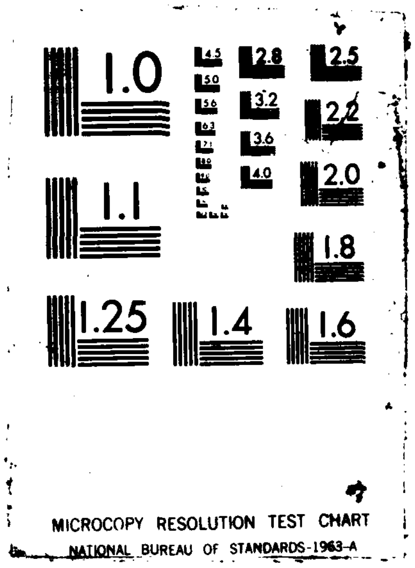

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED

AD-E500 874

Copy 9 of 33 copies

AD-A185 663

(2)

IDA PAPER P-1828

ENVIRONMENT EXTENSIBILITY IMPACT ON THE STARS SEE ARCHITECTURE SEE-ARCH-007-001.0

William E. Riddle
Jack C. Wileden

April 1985

Prepared for

Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, VA 22311

UNCLASSIFIED

IDA Log No. HQ 85-29769

87 10 1 135

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None.
2a SECURITY CLASSIFICATION AUTHORITY 2b DECLASSIFICATION/DOWNGRADING SCHEDULE	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release/distribution unlimited.
4 PERFORMING ORGANIZATION REPORT NUMBER(S) P-1828	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION STARS Joint Program Office 8b OFFICE SYMBOL (if applicable) SJPO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) 1211 Fern Street, C-107 Arlington, VA 22202	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-D5-429 / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
Environment Extensibility Impact on the STARS SEE Architecture SEE-ARCH-007-001.0 (U)

12 PERSONAL AUTHOR(S)
William E. Riddle, Jack C. Wileden

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM ______ TO ______	1985 April	88

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Software Engineering Environment; Extensibility; Software Tools; Environment Architecture; Joint Service Software Engineering Environment (JSSEE); Software Technology for Adaptable, Reliable Systems (STARS).

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This document addresses the topic of environment extensibility as part of an effort to define a Joint Service Software Engineering Environment (JSSEE). The concept of a JSSEE has evolved from work in the Software Engineering Environments area within the Software Technology for Adaptable, Reliable Systems (STARS) program of the U.S. Department of Defense. A JSSEE is intended to be a modern, comprehensive software engineering environment, which implies that it must be capable of gracefully absorbing new software development tools and technology as they become available. This in turn implies that extensibility is a central concern in the definition and devleopment of JSSEE environments. Prelminary issues are treated by providing definitions related to environment extensibility, considering some dimensions of the extensibility question, and framing a statement of the problem addressed. A variety of existing environments are reviewed, architectural considerations are discussed, and strategies for creating and evolving extensible environments are among the subjects.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

IDA PAPER P-1828

ENVIRONMENT EXTENSIBILITY IMPACT ON THE STARS SEE ARCHITECTURE SEE-ARCH-007-001.0

William E. Riddle
Jack C. Wileden

April 1985

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-D5-429

Acknowledgements

We would like to thank the members of the JSSEE Group for their valuable comments on an earlier version of this paper. Special thanks go to Sam Redwine for careful review, insightful observations, incisive questions and many good ideas.

Table of Contents

1. Introduction

This report addresses the topic of environment extensibility as part of an effort to define a Joint Service Software Engineering Environment (JSSEE). The concept of a JSSEE has evolved from work in the Software Engineering Environments area within the Software Technology for Adaptable, Reliable Systems (STARS) Program of the U.S. Department of Defense. A JSSEE is intended to be a modern, comprehensive software engineering environment, which implies that it must be capable of gracefully absorbing new software development tools and technology as they become available. This in turn implies that extensibility is a central concern in the definition and development of JSSEE environments.

In the remainder of this initial section, we treat the necessary preliminary issues by providing definitions related to environment extensibility, considering some dimensions of the extensibility question and framing a statement of the problem we have addressed. Then, in Section 2, we review a variety of existing environments and make observations about the structures and technology that have enhanced the extensibility of these environments. Architectural considerations related to environment extensibility are discussed in Section 3 and strategies for creating and evolving extensible environments are the subject of Section 4. In the conclusion, Section 5, we reiterate the major conclusions and recommendations stemming from this study.

1.1. Definition of Environment Extensibility

In this section we define the concept of environment extensibility and distinguish it from other types of changes to an environment. To do this, we first establish some terminology regarding environments themselves, in order that the distinctions that we wish to make may be as crisp and clear as possible.

A software engineering environment is, for purposes of this report, a computer-based, automated utility to assist in activities associated with the creation and/or evolution (including in-service support) of a software system. It is part of a larger environment that includes unautomated tools supporting software creation and evolution. By 'environment', however, we will mean the automated utility that is the software engineering environment.

An environment is composed of hardware and software. Our interest here is in the software component of that utility, and more particularly that part of the software component which is dedicated to providing the functionality available to the environment's users. We therefore make the following definitions:

DEFN: the 'environment host' is a virtual machine providing the (usually fairly primitive) objects and operations useful in implementing the rest of the environment's software; the environment host typically comprises some software in addition to the underlying hardware upon which the environment as a whole executes

DEFN: the 'environment body' is that software which implements the environment's functionality and is itself implemented upon the environment host

DEFN: the 'environment interface' is the software concerned with delivery of the environment's functionality in an easily usable form {1}; the interface is primarily implemented upon the environment host but may also use some of the facilities provided by the environment body

Our view is depicted in Figure 1 and our focus is on that part of the environment labeled "environment body" in the Figure {2}. By focusing in this way, we have turned our attention away from issues that concern modern operating systems (these would be treated in considering what we have called the environment host) and issues that concern the human engineering of the environment (these would be treated in considering what we have called the environment interface {3}). Instead, we have turned our attention, for the short two-month period of our study, to the central core of the environment to get at the major issues that affect environment extensibility.

An environment architecture is a model that helps in understanding the environment -- Figure 1 gives a simple architecture that helps understand the rough partitioning that focuses our study. The architecture can be a model that helps the users understand the environment's capabilities, but more usually the model is oriented toward construction of the environment, i.e., it is aimed at the environment's builders and/or maintainers rather than its users [12]. For this study, we use the following definition:

{1} An interface is a shared boundary. In our usage, we allow the interface to have a "width" and include software that mediates the flow of information across the boundary.
{2} Although we need to make the distinction between the environment and its body to specify the scope of our work, we will frequently use the word 'environment' to mean 'environment body' except where the distinction needs to be made clear.
{3} Studies of the environment interface, including its extensibility characteristics, have been reported on in [3] and [23].

DEFN: an 'environment architecture' is a construction-oriented model of the environment's organization which defines its major components and the rough logic of their inter-relationships.

Our focus on the environment body means that we are interested in the architecture for this part of an environment.

For the purposes of investigating environment extensibility, we need to distinguish between the information stored in an environment and the processors that work on this information {4}. This leads to the following definitions:

DEFN: an 'information fragment' is a component of an environment body's architecture consisting of some data pertaining to the software system being created or evolved or to the project which is carrying out the creation or evolution {5}

DEFN: a 'tool fragment' is a component of an environment body's architecture that transforms information fragments or creates new ones and contributes to providing some part of the environment's functionality

DEFN: an 'information repository' is the collection of information fragments

DEFN: a 'tool set' is the collection of tool fragments

The information and tool fragments are the fundamental parts of the environment body. In addition to these fundamental parts, an environment body's architecture must specify an infrastructure, defined as follows:

DEFN: an 'infrastructure' is the component of an environment body's architecture that specifies a structure for the information repository and tool set

The infrastructure specifies the ways in which the fragments interrelate and thus captures the organization of the information fragments, the interactions among the tool fragments, and the ways

{4} This distinction may be of more general utility, but we do not wish to argue that here.
{5} We speak as if an environment supports only a single project. We do not preclude the possibility that an environment can simultaneously support several projects -- it's just easier to talk in these simpler terms and it does not restrict the applicability of our conclusions.

in which tool fragments use and produce information fragments {6} -- it provides, in essence, the "glue" that holds the parts together in some logical manner reflecting the processing that must be performed. It implies the existence, in the environment body's implementation, of mechanisms for maintaining the specified information organization (e.g., a database subsystem), supporting the tool fragment interactions (e.g., a process management subsystem), and controlling the utilization and production of information fragments (e.g., an interface to the database system that controls the retrieval and storage of information).

With the preceding definitions regarding environments and their architecture, we can now give a precise statement of what we mean by environment extensibility, distinguish it from other kinds of changes to an environment, and relate it to environment architecture issues.

DEFN: 'Environment extensibility' is the ability to modify an environment in response to changes in the ways that the environment will be used.

Changes in the ways that the environment will be used will necessarily induce changes in the capabilities demanded by the users and these changes may be either additions to, deletions from, or modifications to the environment's functionality. It may be possible to accommodate some of these changes through modifications to the environment interface or environment host -- these are not of concern when considering the architecture of the environment body as in our study. Therefore, we use the following more specific definition:

DEFN: 'environment extensibility' is the ability to modify an environment body in order to satisfy changes in the requirements for the functionality provided at the boundary between the environment body and the environment interface

This version of the definition limits our concern to changes needed to produce changes in the environment's functionality. The scope of our concern for various changes will be discussed more extensively in the following subsection.

A fundamental assumption underlying this study is that the architecture of an environment body can significantly influence the environment's extensibility. A particular architecture may make the changes involved in extending the environment easier or more difficult. The architecture itself may or may not need to be

{6} In general, it might be possible that tool fragments themselves are "consumed" and "produced" or that the infrastructure changes dynamically over time. We do not consider these more complicated situations here.

modified in order to achieve some particular extension of the environment. It is precisely these issues that are the central focus of this report.

1.2. Dimensions of Environment Extensibility and Scope of Our Study

In this section, we indicate various reasons for changing an environment and relate them to changes that both would and would not be considered instances of environment extensibility under our definition. We also consider some dimensions of the extensibility issue, based primarily on properties of the changes that might be made to the environment's functionality.

Changes to an environment will be precipitated by changes external to the environment itself. Some of these external changes are:

-- introducing a new team structure

-- hiring new members to the project team

-- starting a project in a new application area

-- starting a new project in the same application area

-- moving an environment into a new software creation and evolution organization

-- changing the software development or project management method or methodology

-- requiring more extensive automated support for a method or methodology

-- requiring the use of a different language in which to program

-- introducing new documentation requirements

We are interested in external changes such as these only to the extent that they induce changes in the functionality required of an environment. For example, we do not view employing the environment with new personnel or on a new project, when it does not require any modification to the required functionality, as a situation requiring extensibility. Neither do we view minor 'tailoring' or 'customizing' (e.g., by defining abbreviations for command names) as an instance of extensibility. Thus, we are not interested in the root causes of change in this study; rather, we are only interested in the need to provide a change in the environment's functionality for whatever reason.

A subtle, additional restriction follows from this focus on functionality changes without regard to their cause. Without concern for root, external causes, we cannot adequately consider the effect that extensibility may have on the 'world' using an environment. For example, a change in the team structure might induce a change in the management support facilities and accommodating this change might require a change in the management method used for the project. In essence, we assume that any perturbations or inter-linkages such as these are addressed before a functionality change is requested or appropriately handled, prehaps by requesting additional functionality changes, should they occur.

We have also chosen to not consider changes that primarily impact an environment's implementation. Some changes in this category are:

-- moving an environment to a new host

-- enhancing the performance of the environment

-- implementing the environment in a new programming language

We recognize that changes such as these may incur significant work. For example, porting may involve major (say if the move is from a centralized to a distributed host) or minor (say if the move is from one Ada-based host to another) revisions to the environment. But we feel that they rarely, if ever, induce significant changes to an environment's architecture. And so we focus our study, without significant restriction in our minds, by not considering implementationrelated changes; in particular, we do not consider the issue of portability.

The ease of extending an environment is usually the primary concern. But there are other concerns, among them:

-- time and cost: the resources consumed in extending the environment

-- reliability: the extent to which the extended environment will suitably deliver the new or modified functionality and continue to suitably deliver the unmodified functionality

Usually, these concerns will be traded off against each other. For example, it may sometimes be reasonable to enhance the reliability of environment extension even though this may lead to more cost, time, and/or difficulty in performing the extension. While these other concerns are quite important, our study has focused solely on the ease of extension.

Changes to an environment's functionality may have a variety of properties, and it may be more or less difficult to modify an environment depending on what these properties are. Among the properties of a functionality change are:

-- longevity: ease of extension may depend on whether the functionality change is to be temporary or permanent

-- frequency: it may be more difficult to respond to changes that occur frequently

-- compatibility: the constraint that certain compatibility constraints be observed may complicate the process of extension

-- anticipation: it may be easier to accommodate an expected functionality change than one that is completely unanticipated

We have not considered how these various properties affect the ease of extending an environment.

In summary, we have restricted our attention to the ease of responding to changes in the environment's functionality without regard for the specific causes inducing the functionality change or the specific properties of the functionality change.

1.3. Statement of Problem

The objective of this report is to consider the relationship between an environment body's architecture and the ease of responding to changes in the functionality required of that environment. Specifically, we seek to identify aspects of an environment body's architecture which facilitate, inhibit, or are unaffected by a need to respond to changes in an environment's functionality, with the intent of:

-- providing a clearer understanding of architectural considerations related to extensibility, and

-- contributing to the formulation of strategies for building extensible environments.

Our observations and conclusions will be expressed in terms of the tool set, information repository and infrastructure of an environment's body.

2. Survey of Current Environment Extensibility Capabilities

Few, if any, current-day environments have been designed with extensibility in mind. However, several existing environments are extensible to some degree. In fact, of course, all environments are extensible in the sense that, with a sufficiently large expenditure of effort (possibly enough to rebuild the entire environment), any environment can be extended. Our interest here is in the extent to which certain capabilities make it relatively easier to extend the environment.

In this section, we discuss the characteristics of several existing environments that are pertinent to their extensibility. For each environment, we first describe the environment's architecture and then discuss the characteristics that facilitate extension of the environment to meet new environment interface requirements.

In addition to specific environments themselves, extensibilityenhancing technology has been developed outside of the context of any particular environment. Some of this technology has been developed with the intent to use it in environments; some has been developed for more general use. Pieces of this technology pertinent to environment extensibility are also discussed in this section.

Although it had been anticipated that a comprehensive literature search would uncover either horror stories or success stories related to environment extensibility, in fact no such stories were found during our survey. We suspect that this means environment developers and users are reticent to share their experiences, in public at least, rather than that such stories do not exist.

2.1. Extensibility in Current Environments

2.1.1. Toolpack/Odin

The Toolpack environment [21], created by a consortium of university, Government and industrial research labs, is an experimental environment created to support the development of numerical analysis software in Fortran. Toolpack is a rare example of an environment that was designed with extensibility explicitly in mind. As a result, Toolpack's architecture as it relates to extensibility is particularly interesting. The aspects of that architecture most relevant to extensibility are contained in the Odin subsystem.

Toolpack provides its users with a product-oriented interface. Rather than specify what tools should be invoked, a Toolpack user indicates what object (e.g., an object file, an test report, etc.) is desired and, assuming that the object can in fact be produced given the current state (e.g., no necessary inputs are missing), Toolpack carries out the necessary steps to produce the desired object.

Odin, developed at the University of Colorado, is intended to serve as the basis for an extensible program development environment. Odin is an outgrowth of the Toolpack project and an extension of the

Integrated System of Tools (IST) subsystem found in Toolpack [7]. For our purposes here, Odin's major role is to determine how an object requested by a user can be produced and then to cause the necessary steps to be carried out so that the requested object is indeed produced.

2.1.1.1. Architecture of Toolpack/Odin

The Toolpack/Odin architecture consists of a collection of tool fragments and a file system (see Figure 2). The file system is in fact a virtual file system. This means that a specific file need not actually be stored if it can be regenerated from other files using one or more tool fragments. The result is that only a small number out of the many files that a user believes to be in existence may actually be physically present in a Toolpack file system at any given time, but that from the user's point of view they may all be considered to be present there. This situation arises from the approach to defining information fragments and their relationship employed in Toolpack.

The Toolpack/Odin approach is to treat all information fragments as files or sets of files and to provide mechanisms allowing users to define file types and operations specified in terms of those file types. The various file types may be thought of as distinct views of some part of a software system. For example, test.f:fmt and test.f:obj might be two file types corresponding to two different views of the Fortran program stored in the file "test" (the extension .f denotes a Fortran base (primitive) object type in Odin), the former a formatted view of the source and the latter an object code view.

The tool fragments and information fragments, where each information fragment corresponds to a file in the virtual file system, in a Toolpack/Odin system are related through a structure called the dependency graph (or dependency DAG, for directed acyclic graph). A sample dependency graph appears in Figure 3. This graph explicitly represents how every type of information fragment defined in the Toolpack/Odin system can be created from some other type of information fragment(s) through the application of one or more tool fragments. Odin uses this information to determine how a requested information fragment can be created, then invokes the appropriate sequence of tool fragments with the appropriate initial information fragment as input to achieve that result. Naturally, as long as certain types of virtual files are always retained in the physical file system (namely those corresponding to nodes in the dependency graph that have arcs emanating from them but no arcs coming into them) other types of files (namely those corresponding to nodes with at least one arc coming into them) never need to be retained in the physical file system. Those file types that are not retained can always be regenerated if needed, using the information in the dependency graph.

2.1.1.2. Extensibility Aspects of Toolpack/Odin

One important extensibility aspect of the Toolpack/Odin architecture is its use of small granularity tool fragments. This approach of decomposing large tools into their constituent fragments, then composing the appropriate fragments in order to accomplish a particular task, is a significant boon to extensibility. In particular, it encourages re-use of individual fragments and provides building blocks to aid in the creation of new tools.

Toolpack/Odin's use of typed information fragments is also significant from the perspective of extensibility. Although the information fragments tend to be of much larger granularity than the tool fragments, the concept of fragments and the notion of types are both valuable for extensibility, since sharing and re-use of information fragments contributes in the same way as sharing of tool fragments, while a type structure provides some control over use of information fragments and the prospect of some checking for correct usage patterns. Since Odin controls all tool invocations, the Toolpack/Odin system does in fact enforce the restriction that information fragments of a given type are only used by tool fragments that they are intended to be used by.

Two languages are provided by Toolpack/Odin. One is a command language, which permits a user to request the creation of a view (file type or information fragment). Essentially, this language lets a user name a view that is desired (e.g., test.f:obj) and optionally indicate where a copy of that view should be stored (e.g., test.f:obj>test2.f:obj). By hiding the exact nature of how effects are achieved, this language renders the environment's internal structure more malleable while preserving its external appearance to users and thus enhances extensibility.

The other Toolpack/Odin language is a specification language in which users can define new views and indicate how those views are to be created. Odin transforms this information into a derivation graph, which indicates what tools can be used to create a file of one type from a file of some other type. This graph then encodes the relationships among information interfaces and tools, and provides an internal, accessible representation of all the possible file types as well as the kinds of operations that can be performed on them. Although its use in Toolpack/Odin is limited, such internal accessible representations can significantly enhance extensibility.

2.1.2. Ada Environments

Environments supporting the development of Ada programs were defined in the mid-1970's as an outgrowth of the U.S. Department of Defense Ada Program. The initial design and definition efforts culminated in a definition document popularly called Stoneman [6] which appeared in the late-1970's. An Ada Programming Support Environment, as defined in the Stoneman document, provides automated support for the full spectrum of life cycle activities for large, embedded software systems.

Multiple implementation efforts are currently in various stages of completion for several environments supporting Ada programming activities. The Ada Language System (ALS) [25] is being built by SofTech under contract to the US Army. A variation, called the Ada

Language System/Navy (ALS/N) [18], has been specified and is currently under procurement by the US Navy. And the Ada Integrated Environment (AIE) System [13] is being built by Intermetrics under contract to the US Air Force. These current Ada environment implementation efforts have directed most of their attention to the environment capabilities supporting the programming phase of software development.

2.1.2.1. General Architecture of Ada Environments

The original specification of an architecture for environments supporting the development of Ada software systems appeared in the Stoneman document. This architecture, depicted in Figure 4, consists of a hierarchy of virtual machines, with each virtual machine providing a base for implementing the next higher level virtual machine. The most primitive virtual machine, closest to the actual hardware, is called the Kernel Ada Programming Support Environment (KAPSE). It provides basic operating system services including terminal access facilities, editing facilities, Ada runtime support, and data management. The next virtual machine in the hierarchy is called the Minimal Ada Programming Support Environment (MAPSE) and is essentially an Ada virtual machine, providing the capability to prepare, compile and execute Ada programs. The outermost virtual machine is the Ada Programming Support Environment (APSE) itself, providing capabilities supporting the team-based creation and evolution of Ada software according to some general or specific method.

The data management capabilities specified by Stoneman are only minimal extensions of a normal file system. Each information fragment stored within the database is considered to be a relatively large unit of information such as a program, report or collection of test data. Each information fragment has a name by which it can be uniquely retrieved. In addition, attributes in the form ‹key,value› can be used to categorize information fragments for the purposes of retrieval. Some of these attributes are pre-defined but the set of categorizing attributes can be extended as required. The attributes provide a flexible way of imposing a logical organization on the information fragments so they can be retrieved according to criteria that reflect concerns such as version control, documentation, and quality control.

2.1.2.2. Architectures for the Current Implementation Efforts

The current, Government-sponsored Ada environment implementation efforts have adhered, in general terms, to the general architecture specified in Stoneman. The particular architectures refine the general architecture to:

-- reflect the particular ensemble of tools included in the environment,

-- accommodate the particular computer and operating system upon which the environment is hosted,

-- accommodate the specific database organization chosen for the implementation, and

-- provide a usable, Ada-like command language.

The architecture of the ALS, as pictured in Figure 5, is essentially identical to the Stoneman architecture except 1) for the recognition of the KAPSE as being a virtual machine layer built on top of the native operating system, and 2) the lack of a distinction between the MAPSE and APSE virtual machine layers. An alternative view of the architecture is given in Figure 6. This view is more oriented to explaining to environment users the dynamics of system operation under typical scenarios and, as such, distinguishes the database portion of the system as a separate layer which supports the operations carried out by the individual tools.

A major aspect of the ALS effort is the definition of a much richer set of pre-defined information fragment attributes than is laid out in Stoneman. In addition, the ALS database definition explicitly recognizes associations among information fragments as a special type of information fragment attribute having a set of pointers as its value.

Because the ALS/N is a variation of the ALS for specific, Navy target computers, the ALS/N architecture, given in Figure 7, is identical to that of the ALS. The tool layer in the ALS/N provides for additional sets of tools such as ones that provide support for working with MTASS-developed software. In addition, the definition of the ALS/N explicitly defines an architecture for the physically separate part of the environment that runs on the various 16and 32-bit target computers toward which the environment is oriented. The architecture of this Run-time Executive part of the overall environment is pictured in Figure 8.

The architecture for the AIE is given in Figure 9 and is, again, essentially the Stoneman architecture. In the AIE, the database is explicitly defined as a virtual machine layer and this leads to the elaboration, in the AIE, of the KAPSE layer as shown in Figure 9. The structure of the database itself, shown in Figure 10, is considerably more elaborate than specified in Stoneman -- this follows from the more explicit delineation of the tools provided by the AIE and specific attention to the problems of supporting configuration management and version control.

2.1.2.3. The Arcturus Environment

The APSE implementation efforts discussed in the previous subsection have all tended to elaborate the inner virtual machine layers. In another APSE-related effort, the emphasis has been on elaborating the outer layers. This is a research effort being carried out over the last five years at the University of California, Irvine, and has resulted in a prototype system named Arcturus [29].

The Arcturus prototype offers execution with both compiled and interpreted Ada, template-assisted Ada text editing, tools for measuring the performance of Ada programs and displaying the data in easily interpretable form, formatted listing of Ada programs that can be easily controlled to match individual preferences, a program design language compatible with Ada, and assistance for refining designs into executable code. This list of capabilities emphasizes that the focus has been on the user interface and the facilities provided from within the APSE virtual machine layer. This emphasis is reflected in the architecture of the Arcturus system, given in Figure 11, which indicates a refinement of the APSE layer within the Stoneman architecture.

2.1.2.4. Extensibility Aspects of Ada Environments

All of these specific architectures for Ada environments are variations on the theme established in Stoneman. They share the characteristic that the environment is structured as a layer of virtual machines that provide their defined capabilities through the use of the capabilities provided by lower level virtual machines. Our observations about the extensibility aspects of Ada environments are therefore stated with respect to the Stoneman architecture but should be understood to apply to all the specific architectures discussed above.

The definition of the KAPSE layer provides for the portability of an Ada environment -- as such, it does not directly contribute to extensibility as we have defined it for the purposes of this study.

The layered structure of the environment facilitates extension by providing a variety of fixed points that can potentially remain unchanged when modifying the environment in response to a change in the requirements for the environment interface. To provide a new capability at the interface, the hope is that it could be implemented upon the MAPSE using the capabilities of this higher-level virtual machine. Thus, not only would the implementation proceed more quickly because of the high-level language nature of the MAPSE, but the newly created environment would result from the re-use of at least all of the previous MAPSE and KAPSE. For some changes, however, it may be necessary to make changes in the MAPSE or even the KAPSE. Because the expectation is that changes in these deeper layers in the environment would be infrequent, the net result of a layered architecture such as found in Ada environments should be a decrease in the average effort to extend the environment.

Extensibility is also enhanced by the use of attributes to describe information fragments and the fact that the attribute set can be arbitrarily extended. If a change requires new types of information fragments then these can be installed in the environment by the addition of new attributes that reflect these new types of fragments. If the old attributes remain unchanged, then any parts of the system that rely on them can also remain unchanged. If some of the old attributes have to change then only those parts of the system that rely on them will have to be inspected for possible change.

2.1.3. Gandalf

Gandalf is a family of environments, each intended for different audiences, that share a common paradigm as to how software systems should be developed [20]. The paradigm emphasizes the incremental preparation of software by teams of people with the extensive use of automated support stemming from the environment-processable description of project team and software system structures in addition to the description of the software itself.

The Gandalf project was begun at Carnegie-Mellon University in 1978 with the intent of developing an Ada programming support environment which did not necessarily use the layered architectural structure espoused in Stoneman. The focus was on providing support for the programming of software systems by teams and so the emphasis was on supporting programming, configuration management and team interaction activities. Over the years, the emphasis on Ada has lessened, but the environment's scope of activity coverage has remained essentially the same.

2.1.3.1. Architecture of Gandalf Environments

The paradigm supported by Gandalf environments does not require the eved, e.g. "show the value of variable A at this point in the program". The system automatically causes the appropriate actions to be taken to be able to achieve the requested effect. Depending on what changes have been made to the code for the modules, this may involve any or all of recompilation, reconfiguration or re-execution.

Thus a user's view is that a Gandalf environment is a monolithic system that does not have tools or collections of tools in the traditional sense. Most of the literature on Gandalf emphasizes this view and none of it provides an architectural diagram per se. From the literature discussing the ways in which Gandalf capabilities are implemented, the architectural view given in Figure 12 can be inferred but the suitability of this architecture has not been verified.

2.1.3.2. Extensibility Aspects of Gandalf Environments

Because of the uncertainty as to the exact architectural structure of Gandalf environments, our observations pertain to enhancing extensibility by hiding the environment's actual architecture from its sers, providing the capability to (semi-)automatically generate parts of the environment, and using declarative descriptions for the activities supported by the environment. These contributors to environment extensibility appear more extensively in the Gandalf work because of its concern for the development of a family of environments.

One hallmark of Gandalf environments is the separation of concern for how to develop software from concern for how individual tools are used to support this development. The monolithic interface to a Gandalf environment does not reflect a traditional set of tools that relate to the usually recognized steps such as compilation. In addition, the interface constrains users to think and work directly in terms of the system's description (that is, the system's code). Thus the interface reflects the paradigm used for software development and the representation of the system but does not fix the tool set provided for following the paradigm or manipulating the descriptions. This enhances the environment's extensibility because the exact nature of how various effects are achieved is hidden from the users and therefore does not have to be preserved when making a change in the environment's design or implementation.

Another hallmark of the Gandalf work is use of well-defined languages to capture both information about the system being developed (in addition to information about its operation) and information about the nature of the project itself. This information is traditionally held outside the environment (if it is recorded anywhere) but is needed in Gandalf environments to support the automatic performance of activities such as change control and system regeneration. These languages enhance extensibility because the environment itself is prepared to interpret and follow the directives that are encoded in these languages and can therefore support new ways of doing business as long as these new ways can be described.

This aspect of the Gandalf project tends to lead to environments that exhibit the ultimate in extensibility, namely, the environments do not have to be changed at all in response to changes in the way of doing the business of software creation or evolution. As long as the languages are general enough, then the result of any changes in the way of creating or evolving software can be described using the languages and the system, given the new description, can support the new approach without change. Rarely, however, can the languages be defined to be general enough. But it is useful to head in this direction and separate the specification of what is to be done from the mechanism of performing it through the use of well-defined languages for describing software and project characteristics.

The Gandalf project has also extensively relied on the automatic generation of processors for descriptions in the languages. This means that changes (reflected in changes in descriptions in the languages) can be fairly easily accommodated because the processors can be automatically changed. This enhances the extensibility of an environment because changing the environment can be easily done as long as the new requirement for the environment can be captured as a change in the descriptions in the languages.

2.1.4. Unix

The Unix operating system [17] was initially developed, at Bell Telephone Laboratories, in the very late 1960's in an attempt to provide the same services on small computers found in modern operating systems for large mainframe computers. Over the years, its popularity has spread, particularly within the academic research community, and it is currently widely available and a major operating system for 32bit micro-computers.

Viewed as an environment, Unix is a loosely integrated collection of tools supporting a wide variety of approaches to software creation and evolution. Many of the tools are simple ones that perform text transformations or processing of general utility. There are also many more-powerful, high-level tools for specific tasks, such as document preparation, found during software development and maintenance. Few, if any, of the tools are wedded to a particular way of performing software creation and evolution -- the tools are, for the most part, general-purpose ones that can be used in a variety of ways to support a variety of methodologies.

2.1.4.1. Architecture of the Unix Environment

The Unix operating system has a very simple layered structure, pictured in Figure 13. At the core of the system is a kernel operating system providing facilities for process management and the transfer of information between processes, among other things. The rest of the system is a collection of tools that run on the virtual machine provided by this kernel. A command language processor, which is just one of the tools, provides a capability to write programs controlling tool invocation (including the recursive invocation of the command language processor itself).

Unix provides a more or less standard file system for the storage of information fragments. The file system is hierarchical with file names indicating the directory path needed to access a file. Some use is made of naming conventions to determine which files or tools to use in response to a user's command, but this automatic interpretation of file names is rare. Users may establish naming conventions to aid in the categorization of information fragments, but the system provides no direct help in interpreting file names for the purpose of storing information fragments into and retrieving information fragments from an information repository.

Viewed statically, the collection of tools has no structure (although one could view the operating system services as a lower level collection of tools). This "flat" collection of tools can be dynamically structured (and re-structured) into any general network structure needed to perform the necessary processing.

Partially this network structure can be developed using the ability to call any tool from within any other tool. This capability allows rather general, hierarchical structures of tools to be created.

The more general means of developing an arbitrary network structure is to use the Unix capability to "pipe" information from one tool to another without having to build temporary files in which to store the information. This allows the tools to be rather freely cascaded together in whatever way is required as long as the needed structure is a simple sequence of tools. Some versions of Unix provide the capability to create more general tool sequencing structures in which the output of one tool may be directed, as input, to several other tools.

In sum, the architecture of Unix itself defines that the tool collection is a flat collection of tools and the means exist for this tool collection to be dynamically structured in rather arbitrary ways.

2.1.4.2. Extensibility Aspects of the Unix Environment

As in Ada environments, the innermost virtual machine layer is primarily an assist for portability and therefore of little assistance for extensibility as we have defined that architectural issue.

However, the Unix kernel has some facilities that are useful in extending the Unix environment. Many of the traditionally immutable parts of a kernel operating system are, in the Unix kernel, defined via user accessible and changeable data structures such as tables. (One notable example is the definition of the characteristics of terminals used for interactive access.) Thus, these aspects can be freely changed to meet new requirements that end up requiring changes to the kernel in order to accommodate. This is another example of technique for facilitating extensibility that appears in the Gandalf environments -- separate definition of the data controlling some activity and use of an interpreter of this data so that new requirements can be accommodated by changing the data rather than changing the processing of the data.

Extensibility of the Unix environment is also facilitated by the fact that a small number of "standard" kernels have evolved over time. These kernels have been propagated fairly extensively throughout the community with the result that tool development can proceed independently and in parallel at various Unix sites. As a result, there is a higher probability that a need for a new tool to meet some new environment interface requirement may be met by looking to other parts of the Unix community and importing the tool from some other location.

Enhancing this support for extensibility is the practice within the Unix community to prepare small tool "pieces", rather than large, monolithic tools, that can be cascaded or otherwise combined to provide some needed high-level capability. This (almost extreme) decomposition of the tools into small fragments increases the probability that a tool fragment will be found when a need for it arises to accommodate some change in the requirements for the environment interface. (It should be noted that the efficient support for combining tool fragments, provided by the Unix kernel, is critical to the success of this assistance to environment extensibility.)

Another extensibility aspect of this high granularity decomposition of tools along with support for the efficient combination of tool fragments is that it encourages and assists a prototyping approach to tool development. With the large number of generalpurpose tool fragments that have evolved over time, it is fairly easy to quickly generate functionally accurate but perhaps inefficient versions of a newly needed tool -- usually this can be done in a matter of days for even rather complicated tools. Thus a request for a new environment interface can be accommodated, to some degree at least, very quickly. This has the usual effect of permitting the exact nature of the requirements for the tool to evolve through use of prototype versions. It also has the effect of being able to extend the environment quickly while letting the performance or capability characteristics of the extension reach an acceptable level more slowly.

This is really just a particular effect of the fact that the tool fragments in Unix are set up to be dynamically and freely combined in whatever way might be needed to provide capabilities at the interface. The fact that it can be done quickly and lead to a functionally accurate but perhaps inefficient result aids prototyping. The fact that it can be done at all provides extensive support for extensibility. This is because it assists the implementation of new tools by re-using existing tool fragments along with newly created ones.

2.1.5. Knowledge-Based Software Assistant

The Knowledge-Based Software Assistant (KBSA) is a research project aimed at producing a software development environment supporting a knowledgebased perspective on the software development and maintenance process. The intent is to "introduce a fundamental change in the software lifecycle maintenance and evolution occur by modifying the specifications and then re-deriving the implementation rather than attempting to directly modify the optimized implementation." The goal is to have all software development and maintenance activities carried out at the specifications and requirements level, not the implementation level. "The transformation from requirements to specification to implementation will be carried out with automated, knowledge-based assistance." KBSA was defined in a report produced by Kestrel Institute for Rome Air Development Center in 1983 [11].

The KBSA itself is intended to "put the machine in the loop", i.e., have all activities carried out during software development and maintenance mediated by the KBSA. The result would be an environment that would monitor, capture and reason about those activities and serve as a knowledgeable assistant to the human software developer.

2.1.5.1. Architecture of KBSA

As indicated in Figure 14, the KBSA architecture is intended to be centered around a core called the 'framework'. The framework will coordinate and direct all activities in the environment. It contains the activities coordinator, which makes the actual decisions regarding KBSA activities and the knowledge-base manager, which provides the coordinator with information about the software development process and its current status as well as managing all the information fragments related to the software project under development. All agents, human and mechanized, involved in the project would be known to the activities coordinator, whose role is to enforce policies and procedures governing their actions. The policies and procedures, in turn, would be described to the activities coordinator via a language, and that information stored by and accessed via the knowledge-base manager.

The framework in the KBSA architecture coordinates the activities of a set of "facets", which may be thought of as sets of tool fragments applicable to various lifecycle activities, a set of project management tools and a support system comprising certain fundamental environment utilities such as version control and an inference engine. The user interface is also considered part of the support system in the KBSA architecture.

2.1.5.2. Extensibility Aspects of the KBSA Architecture

The major extensibility aspect of the KBSA architecture is the activities coordinator. The fact that the activities coordinator can accept, store and reason about knowledge regarding the software development process has significant implications for extensibility. It means that, in principle, a modification to the methodology being employed in a project utilizing the KBSA could be accomplished merely by informing the activities coordinator, using the language provided for that purpose. The activities coordinator would then use its knowledge of the process and of the agents available in the environment to implement the revised methodology. Moreover, it suggests that a change in the set of agents (e.g., addition of new tools with new capabilities) would simply require informing the activities coordinator, which would then incorporate that knowledge into its knowledge base for future reference. Situations in which the new agents might be useful would subsequently be recognized by the activities coordinator when they arose, and the coordinator would call the agents into play at the appropriate times.

2.1.6. Smalltalk-80

The Smalltalk-80 [14] system was developed by the Learning Research Group at Xerox Palo Alto Research Center. Smalltalk-80 combines a programming language and a programming environment, and the two are highly intertwined.

The fundamental viewpoint of Smalltalk-80 is object-oriented. Smalltalk-80 programs, and similarly the Smalltalk-80 system, are composed entirely of objects which send messages to one another. A message may ask an object to perform one of its methods, which is an operation that the object knows how to carry out. Each object is an instance of some class of objects, and all objects in a class have exactly the same set of methods defined for them. Thus the set of methods defined for a class of objects strictly circumscribes the ways in which any instance of the class may be manipulated, since objects ignore any messages asking them to perform any method that is not defined for the class to which they belong. An added richness in this object class structure is achieved through the notion of subclass and inheritance. If a class of objects is defined to be a subclass of some other class, then the subclass inherits all of the methods defined for that other class.

2.1.6.1. Architecture of Smalltalk-80

The Smalltalk-80 system consists entirely of classes and objects. A Smalltalk-80 software developer creates a software system by defining new classes and objects, often as subclasses of pre-existing classes found in the Smalltalk-80 system. In a similar fashion, an environment can be fashioned by creating a set of useful classes and objects.

The actual implementation of Smalltalk-80 is divided into two parts. The Smalltalk-80 Virtual Image constitutes most of the Smalltalk-80 system, including the compiler, debugger, editors, decompiler and file system. This is itself written in Smalltalk-80. The Smalltalk-80 Virtual Machine provides a host on which the Virtual Image can be executed. The virtual machine, which is relatively small, must be written in some language already available on the host computer. The gross architecture of the Smalltalk-80 system is therefore a simple two-level hierarchy of virtual machines.

2.1.6.2. Extensibility Aspects of the Smalltalk-80 Architecture

Smalltalk-80 is a thoroughly object-oriented system. This facilitates extensibility by making modification of the environment and its components much easier to carry out. The Smalltalk-80 style of programming encourages prototyping, which also contributes to extensibility. Finally, the layered virtual machine structure of the Smalltalk-80 system may be seen as facilitating extensibility.

2.1.7. Joseph

The Joseph environment [22] was designed and partially implemented at Cray Laboratories in the very early 1980's. It was developed to support the creation of kernel systems software for high-performance, multipleprocessor computing facilities. It was specifically intended to be developed in parallel with its use, primarily because it was required that the effort to develop Joseph be amortized over the lifetime of the kernel systems software development project which it supported. Joseph was therefore intended to be a highly extensible environment.

Cray Laboratories was closed by its parent company in 1982 and so a full implementation of the Joseph system was never achieved. Nonetheless, a base for the Joseph environment was constructed and tools assisting the description of kernel systems software requirements and design were installed on top of this base. Analysis tools for reasoning about requirements and designs were not developed. The environment was used to develop the requirements and preliminary design for a multiple-processor kernel operating system.

2.1.7.1. Architecture of the Joseph Environment

Joseph has a layered architecture (see Figure 15) with each layer being a virtual machine supporting the implementation of higher layers. None of the layers completely encapsulates the lower layers -- the user of any particular layer can get to it and any of the layers underneath it.

The Unix operating system is used as the lowest layer in

Joseph. It provides basic operating system services and the full complement of tools available within the Unix environment.

The next outermost layer in Joseph provides a flexible, primitive information repository. It is assumed that the information fragments stored in this repository are generally small, on the order of 20-200 characters. Support is provided for the storage of larger information fragments as files, but this was never needed in developing the rest of the Joseph environment. (This capability probably would have been used when tools supporting implementation were added to Joseph.) Each fragment can be annotated with an arbitrary number of ‹key,value› attributes and these attributes are used to both catalogue the information fragments and retrieve them. The query language is a generalpurpose information retrieval one in which the user can ask for fragments satisfying a relatively general criterion stated in terms of specific, string-match, or numeric-range values for various keys. If the pre-established ways of stating queries proves insufficient, then the user my prepare a program that performs the selection based on attribute values and cause this program to be used as part of the retrieval process.

The outermost layer of Joseph provides multiple sets of tools, one set for each major phase of development. The tools available in each set are similar in purpose and implementation and are tabledriven to allow them to be re-used for the differing languages supporting development during the differing phases. Thus there is really one set of tools and different tables are used to make the tools support different life cycle phases. Each of the tools is highly decomposed to allow for multiple use of common tool fragments. This decomposition also allows the efficient delivery of various capabilities because it was found that some capabilities could be provided by utilizing just a few of the tool fragments rather than an entire tool itself. The decomposition of tools was guided by the desire to have common parts and the desire to find more primitive tools that could, by themselves, provide some of the capabilities requested for the environment interface.

2.1.7.2. Extensibility Aspects of the Joseph Environment

The layered structure of Joseph facilitated extensibility by providing fixed "machines" that were general enough to support the implementation of new capabilities. Providing an information repository "machine" was particularly helpful in that it provided the means to organize and freely retrieve highly decomposed structures of information.

The decomposition of information into small fragments supported extensibility by allowing the easier handling of new documents. The use of a very general attribute scheme for categorization of fragments meant that new documents could always be defined in terms of some retrieval criteria and an associated tool for formatting and displaying the retrieved fragments.

The decomposition of the tools into general-purpose tool fragments, along with the tendency to make the tool fragments tabledriven as much as possible, also facilitated extensibility. Implementation of a new capability, such as the capability to process descriptions pertinent to an additional life cycle phase, generally required just the development of a new table reflecting the new languages.

Extensions requiring new tools, rather than just new tables, were also easy to make. Partially this was because of the general support that the Unix "philosophy" (a large collection of relatively small, general tool fragments) provides for expanding the set of tools. It was also facilitated by carefully defining the table formats and contents in a way which facilitated the use of the tool fragments available in and added to Unix.

2.1.8. Distributed Computing Development System

The Distributed Computing Development System (DCDS) [2] has been developed over the last decade by TRW Defense Systems Group, Huntsville, Alabama, under contract to the Army Ballistic Missile Defense Advanced Technology Center. In its original form, called Software Requirements Engineering Methodology (SREM), it was intended to support the definition and analysis of software requirements. This original system has been extended over time to cover the activities of system requirements definition, software high-level and detailed design, and allocation of modules in a software system to the physical processors in a distributed computing facility. The system is compatible with a variety of programming languages and emphasizes no programming language in particular.

DCDS is, therefore, an environment supporting a wide spectrum of life cycle activities (from system requirements analysis through detailed design) during the creation and evolution of distributed, concurrent software systems. DCDS provides a variety of languages for describing software during the various life cycle phases. It also provides a variety of analyzers for reasoning about software specifications and the system they describe. Some support is provided for moving between phases by deriving new descriptions from old or tracing back a sequence of descriptions, but this is largely left to developer intuition, experience, and expertise.

2.1.8.1. Architecture of DCDS

The architecture of DCDS is depicted in Figures 16 through 18. The static view provided by the first three of these views is similar to the architectures given previously -- it reflects a virtual machine hierarchy that progresses from the outermost command language interpreter "machine" to the innermost machine which is the environment host. The more dynamics-oriented view in the last of these Figures indicates the more network-like structure that stems from considering the flow of information within the system.

The central part of the DCDS system is the Extended REquirements Validation System (EREVS). EREVS provides the tools for entry and retrieval of information into the database and derivation of information useful in analyzing a software system for suitability during its creation or evolution. EREVS is an enhancement of the analogous REVS system that was provided within the SREM system and has a layered structure as depicted in Figure 19.

The software description languages provided by DCDS are all relational in nature. A software system is considered to be a collection of entities that have attributes and are related to each other -- two modules in a communications system might, for example, have attributes that indicate their status (ready, full, etc.) and a "sends message to" relationship between them. A variety of relationships are predefined and users may extend the set of attributes with others of their choice.

The analysis capabilities include the ability to "trace" relationships to find interactions or relations between parts which are not directly related, the ability to perform various static analyses on the flow of data within a system, and the ability to perform simulations of partially designed systems. These analysis capabilities are provided by EREVS in response to user requests through the user interface.

The database within the DCDS system is a fairly standard one except that, for efficiency reasons, large blocks of text are stored separately from attribute and relation information. One could consider the database to logically be a traditional file system on top of which has been added a processor of attributes and relations providing the capability to use this information in entering and retrieving blocks of information stored in the files.

2.1.8.2. Extensibility Aspects of DCDS

The software description languages provided by DCDS are based on a variety of formal models of computation. But they all rely on an entity-attribute-relation notation for stating the software models. Because the languages are notationally similar, the EREVS analyzers provide a single set of tools supporting the differing reasoning activities occurring during different life cycle phases.

DCDS can be fairly easily extended to support new ways of performing software creation or evolution, or new aspects of some particular way, by either extending the languages themselves or by creating new languages that use the entity-attribute-relation notational mechanism common to the existing languages. When extension is done in this way, the analyzers are able to work on descriptions in the new languages to support any newly required reasoning capabilities.

The languages currently defined for DCDS already have the ability for users to define the syntax and semantics of new constructs as long as these new constructs adhere to the underlying entity-attributerelation notational mechanism. Thus the languages already support the dynamic extension of the system to cover the description of new characteristics of a software system and the analysis of these aspects.

The evolution of the DCDS system relied upon the ability to cover new life cycle phases by creating new languages appropriate to the new phases but utilizing the common entity-attribute-relation notational mechanism. SREM covered only the software requirements phase of development through the use of a language that employed an entityattribute-relation notation to describe finite-state models of the software's user interface. Analyzers were developed to allow reasoning about the requirements through operations such as tracing a sequence of relations or analyzing for the presense or absence of specific relations. Extension to cover system requirements definition was accomplished by defining a new entity-attribute-relation-based language permitting the capture of functional decomposition models of systems. And extension to cover software design was accomplished by defining additional entity-attribute-relation-based languages permitting the capture of network communication models of the software's logical and physical structure. At each of these extension steps, the already present analyzers could immediately be used to support at least some of the required reasoning capabilities.

The DCDS experience indicates that a fruitful approach to extensibility is to couch all description languages in terms of a common, integrating notational mechanism. This admits the possibility of reuse of analysis tools to support new reasoning requirements. The generalization of this approach would be to have a common internal representation for the variety of languages available to environment users. In such a case, the environment could be extended to support the capabilities reflected in a new language by creating the tools which transform descriptions in the language into the common internal representation and the tools which perform reasoning about descriptions in the new language in terms of the existing tools which support reasoning about descriptions in the common internal representation. In essence, DCDS has used this common representation (that is, the entity-attribute-relation notation) at the user interface and required users to transform their various models into this representation.

2.1.9. FASP

The Facility for Automated Software Production (FASP) has been developed by the Advanced Software Technology Division of the Naval Air Development Center, Warminster, Pennsylvania. Its original release was in 1975 and provided a batch capability supporting development for the Advanced Signal Processor. Since this initial release, it has evolved to provide extensive interactive and batch, life cycle, programming and management support for projects producing Navy standard software written in a variety of languages and intended to run on one of a variety of target computers.

2.1.9.1. Architecture of the FASP System

The architecture of the FASP system [16] is pictured in Figure 20. The unshaded areas of this Figure indicate those parts of FASP that are common to all instances. The shaded areas reflect parts that are particular to a specific instance of FASP, providing the capabilities to work with a specific language and prepare software for a specific target computer. (The Figure indicates that FASP is hosted on a CDC Cyber. While this is true for most instances, versions are available which run on a VAX 11/780.)

The heart of the FASP environment is the Procedure Level which acts as an intermediary between the users and the operating system. Users specify operations in terms of a keyword-plus-parameters command language. These commands are converted into a sequence of operating system directives by the procedure level, which also oversees the flow of data during the execution of these operating system commands. The procedure level also checks for legality of commands, thereby being able to enforce methodological rules and guidelines.

The FASP database provides a hierarchical file system. A userinvisible part of the database provides the definitions of the operating system "programs" needed to respond to user directives in the command language. Users can prepare their own "programs" of commands and have these "programs" interpreted by the mechanisms in the procedure level.

2.1.9.2. Extensibility Aspects of the FASP System

The FASP environment is also an instance of architecting an environment as a hierarchy of virtual machines. As such, it supports extensibility in many of the same ways as the other environments discussed previously.

Adding a new tool to FASP involves making it a part of the inner layers of the environment and then adding the appropriate procedures to allow the tool's invocation in response to user commands. Thus, changes to the capabilities at the environment interface can be accommodated by acquiring or building the appropriate, required, new tool fragments, installing them in the inner portions of FASP, and preparing procedures that deliver the new capabilities by invoking the appropriate tool fragments.

Extensions to data storage capabilities are straightforwardly handled as long as they meet the file-level granularity of the FASP database system. Changes at a higher level (such as the handling of groups of files) or that extend below this level can be accommodated to the extent that software can be prepared to carry out any necessary processing and installed as procedure-reachable tools in the FASP environment.

The inclusion of a procedure level within the environment is therefore the primary support for extensibility provided by the FASP environment. The extent of this support in various situations depends on the extent to which a change can be accommodated by developing new tool fragments as needed, capturing any information fragment changes through tools that work on files, and preparing the procedure(s) which control the invocation of tool fragments to deliver the newly required capabilities to the user.

Extensibility in FASP environments is also enhanced by the conscious decision to provide common tool fragments as much as possible. This has stemmed from desires to be able to port the environment and to be able to support a variety of programming languages and target systems. But, it means that there is a collection of generalized tool fragments that can enhance the ability to provide new capability by the re-use of existing tool fragments.

2.2. Other Extensibility-related Capabilities

2.2.1. Interface Contracts

Imperial Software Technology, Ltd. (IST) has developed the concept of interface contracts as a structuring mechanism for software development environments. As reported by Vic Stenning of IST in an invited talk at the 1984 Ada Applications and Environments Conference, this model structures the interactions among components of an environment by mimicking the contractual process found in the business world. Each component has a technical specification, a management specification and a set of acceptance criteria, and each is responsible for producing some specified deliverables. The environment is then organized as a hierarchy of contracts and projects are decomposed by contracts (and subcontracts), each having its own contract database. Unfortunately, the details of the interface contract mechanism are proprietary, so little more is known about it.

The interface contract mechanism could contribute to extensibility by supporting well-defined interfaces in an environment, which in turn eases modification to the environment and its components.

2.2.2. Common APSE Interface Set

The Common APSE Interface Set (CAIS) has been defined over the last four years in an attempt to establish a basis for the transportability of tools and interoperability of data across Ada environments [1]. In essence, it refines and elaborates the rough definition of a Kernel Ada Programming Support Environment appearing in the Stoneman document [6].

The CAIS provides low-level facilities for both tool implementation and the structuring, management and organization of data. The tool implementation support is in the form of a virtual machine that supports sets of interacting processes. The data structuring, management and organization support is in the form of an entity/attribute "database" that allows working with large collections of information fragments of varying sizes, constructing larger collections of these fragments, and performing the configuration management needed as the contents of these fragments and collections evolve over time.

CAIS would contribute to the extensibility of an environment by providing a fairly high-level, "fixed" base for the environment. The relatively high level of the CAIS capabilities would enhance the ability to add new tools and information fragments with less work than would be required on a more primitive virtual machine. The possibility that a variety of environments will use the CAIS as their host

will increase the probability that newly required tool fragments will already exist and need only be imported to aid in providing newly required capabilities. In sum, CAIS reinforces the enhancement of extensibility to be found in environments with layered architectures by providing a relatively sophisticated base to serve as the an environment host or as the inner layer of the environment's implementation.

2.2.3. Diana and IDL

Diana, a Descriptive Intermediate Attributed Notation for Ada, was developed by researchers from Carnegie-Mellon University, the University of Karlsruhe and Tartan Laboratories [10]. It is based on two earlier proposals for intermediate forms for Ada -- TCOL (or more precisely, the Ada version of TCOL), developed by the PQCC project at Carnegie-Mellon, and AIDA, developed at the University of Karlsruhe.

Diana is intended as an intermediate representation for Ada programs. As such, it represents a natural collection of information fragments for communication between the front-end and back-end of a compiler. It is also intended, however, to be useful as way of communicating among other tools in an Ada software development environment. Diana is designed to encode the information about an Ada program that can be derived from lexical, syntactic and static semantic analyses, but not to contain information resulting from dynamic semantic analysis, optimization or code generation.

Diana is most properly viewed as an abstract data type, defining a class of information fragments. Any instance of that abstract data type, i.e., any specific information fragment, is a Diana representation of a specific Ada program. Diana itself defines a set of operations providing the only means for accessing or modifying an instance of the abstract data type. Although the Diana reference manual offers example implementations for the abstract data type, these only constitute constructive proofs that Diana can be implemented and do not serve to define what Diana actually is.

The concept of an attributed tree serves as a conceptual model for Diana. That is, a Diana representation of an Ada program may be thought of (though it need not be implemented as) a tree of nodes, each of which may have a set of attributes associated with it. Therefore, the definition of Diana is given in terms of a set of classes of nodes and the attributes associated with each class. Conceptually, at least, all information about an Ada program's syntax and static semantics is captured in this attributed tree.

By serving to provide an object-oriented approach to information handling in a software development environment, Diana contributes to the potential extensibility of environments. The object-oriented view isolates tools from the details of the implementation of information fragments and thus facilitates modifications to the environment.

The Interface Description Language, IDL, was developed at CarnegieMellon and employed by the developers of Diana ([15], [19]).

IDL provides a notation in which abstract descriptions of a class of data objects, especially information interfaces, can be formulated.

IDL is especially well-suited for describing information fragments used by tools in a software development environment since it treats all data objects as (possibly degenerate) attributed trees. The main descriptive capabilities of IDL are aimed at defining the various classes of nodes and their associated attributes comprising tions for the same information. Therefore, IDL concentrates on supporting abstract descriptions of information fragments rather than description of the implementation details related to those fragments. IDL does, however, permit specification of some implementation details through a 'representation specification' feature similar to Ada's.

Associated with IDL is a processor for translating IDL definitions of information fragments into implementations that can be used by the tools in a software development environment. This processor assumes a somewhat restrictive structure for tools and their interactions, but given that structure it is sufficient to automate the process of going from an IDL description to the data structure definitions and code necessary to implement the fragment defined by that description.

Like Diana, IDL contributes to the object-oriented approach to information handling in a software development environment, since it offers a mechanism for defining abstract data types and generating instances of those types. Moreover, IDL facilitates a rapidprototyping approach to environment development by making it easy to generate the information fragments that are components of an environment. Both of these facets of IDL make it valuable for extensibility.

2.2.4. Precise Interface Control

The PIC language features and toolset are an outgrowth of the Precise Interface Control (PIC) project at the University of Massachusetts [30]. These PIC facilities are intended to support programming-in-the-large activities. The language features provide a means for expressing module interconnection descriptions while the toolset allows users to obtain interface consistency reports. Module interconnection descriptions contain information about the access that a given module requests to other modules and the access to its own contents that the module is willing to grant to other modules. Interface consistency reports indicate whether the requests and grants of access that have been made by various modules are harmonious or conflict.

The PIC language features and toolset support extremely precise description and enforcement of interfaces. If used in building an

environment, they could contribute to its extensibility by supporting well-defined interfaces among the environment's constituent tool fragments and information fragments, which would greatly facilitate modifications to the environment and its components.

3. Architectures for Extensible Environments

3.1. Required Capabilities

3.1.1. Synopsis of Current Capabilities

Section 2 presented a range of examples of environments and extensibility-related capabilities. Here we distill out the salient features of existing environments that assist extensibility. Thus the following list provides a condensed overview of the extensibility aspects discussed in the previous section, highlighting the various techniques and approaches that are currently being used to support environment extensibility.

3.1.1.1. Decomposition of tools and information into fragments

Toolpack, Gandalf, Unix and Joseph all illustrate the extensibility benefits resulting from an architecture based on flexible composition and re-use of tool and information fragments rather than inflexible, monolithic tools or units of information. (Recall that for Gandalf the user's view is of a monolithic environment but the system itself is decomposed into fragments.)

3.1.1.2. Small granularity of tool and information fragments

Toolpack and Unix are especially good examples of the extensibility benefits that result from keeping the tool and information fragments that compose an environment small, thereby greatly increasing flexibility.

3.1.1.3. Layers of virtual machines

All of the architectures surveyed had some sort of layered structure. The architectures of the Stoneman APSE and Joseph provide specific illustration of how an architecture organized as multiple layers of virtual machines can facilitate extensibility, by providing well-defined 'invariants' in the environment structure to assist in making changes and by providing a set of general purpose, high-level utilities that can contribute to tool-building and enhancement.

3.1.1.4. Object-oriented information handling

Toolpack and Diana/IDL provide approximations, but Smalltalk-80 is the best example of the extensibility benefits arising from the objectoriented approach. The inheritance concept of Smalltalk-80 is particularly valuable for facilitating extension.

3.1.1.5. Productor result-oriented user interface

Both Toolpack and Gandalf hide the tool activation view of their operation from the user, providing instead a declarative, productor result-oriented view in which the user specifies what is wanted rather than how to produce it. This facilitates extensibility by retaining total freedom to modify the tools and activation patterns of the environment's implementation without altering the user's view.

3.1.1.6. Internal, Accessible Representation of Environment Activities

Toolpack offers a first step in this direction through its provision of the dependency graph. KBSA carries it to a much richer, more extensive, more flexible and potentially more valuable level.

3.1.1.7. Well-defined interfaces among tools and information fragments

Toolpack provides a rudimentary version of well-defined interfaces, through its use of file extensions and the dependency graph. IST's contract model goes further toward establishing a stereotyped, homogeneous interface among environment components. The PIC constructs and tools offer more precise specification of interface relationships and capabilities for checking the consistency of those specifications. Diana and IDL provide an abstract information interface and a means for defining other such interfaces, respectively.

3.1.1.8. Isolation/Independence among tools and information fragments

All of the tools and capabilities that contribute to well-defined interfaces concomitantly contribute to isolation and independence. The PIC constructs and tools go a bit further by providing a means for precisely limiting the availability of tools and information fragments so as to both maximize and make explicitly visible their isolation and independence.

3.1.1.9. Environment-oriented application generators

Gandalf and IDL are both examples of application generator technology applied to environments.

3.1.1.10. Support for environment prototyping

The application generators associated with Gandalf and IDL provide support for prototyping of environments. Unix, due to the ease with which tools can be integrated into its pipe-based structure, can also be seen as supportive of environment prototyping.

3.1.1.11. Common Basis for Languages and Analyzers

The common use of the element-relation-attribute framework throughout DCDS, and the table-driven re-use of tools in both DCDS and Joseph, are examples.

3.1.1.12. Extensible Languages

The languages used in DCDS are explicitly extensible. Stoneman APSE environments and Joseph both employ a database model which can be seen as having an extensible query language.

3.1.1.13. Standardized Core

Unix, through its kernel, and the Stoneman APSE environments, through the CAIS, provide standard cores on which additional environment capabilities may be relatively straightforward to construct.

3.1.2. Missing Capabilities

If distilling out common approaches to extensibility based on a set of examples is difficult, identifying missing capabilities on that basis is even harder. In the short time available for this study, no obvious, glaring lacunas were discovered. Many of the capabilities that do exist could be provided in better or more mature ways, and further important capabilities may well be identified in time. But while the need for further exploration and experimentation is clear, obvious shortcomings in currently available set of capabilities for supporting extensibility are not.

3.2. Architectural Principles and Guidelines

The above survey of existing approaches to environment extensibility suggests some general principles and guidelines that should govern the development of architectures for extensible environments. These principles and guidelines are discussed in this section.

The first eight principles and guidelines concern properties that the architecture should possess. Some of these pertain to the overall structure of the architecture. Some pertain to the nature of the information and tool fragments defined by the architecture.

PG #1: The architecture should provide a layer of object types and operations that is intermediate to the defined environment interface and environment host layers.

-- this amounts to decomposition of the object types and operations provided by the environment into ones that are useful in implementing the environment capabilities

-- this intermediate level should probably be actually implemented

-- the intermediate layer provides an opportunity for standardization at a level above the operating system level

-- it also provides the opportunity, if it were adopted for other efforts, to absorb the work of others into the environment

PG #2: The architecture should provide several layers.

-- each will provide a potential invariant in the environment's structure to facilitate changes

-- the interests of extensibility are best served if the layering is 'strict', i.e., if each layer is implemented entirely in terms of capabilities provided by the next lower layer and does not directly refer to any capability provided by any layers below that one

-- the interests of efficiency are best served if the layering is not 'strict', since a strict layering generally introduces high overhead from superfluous context switching; of course, this efficiency penalty can be minimized or even eliminated by suitably intelligent support for the implementation language (e.g., optimizing compilers)

PG #3: The interfaces to the information and tool fragments should be well-defined.

-- this enhances ease of modification

-- these should be machine processable so that reasoning about interfaces and interactions can be supported by automated tools

PG #4: The interfaces to the information and tool fragments should be general.

-- this enhances the opportunity for multiple use or re-use

PG #5: Tools and information fragments should be as independent and isolated from one another as possible.

-- the interactions should be through well-defined agreements (interfaces)

PG #6: The definition of the architecture should separate out interpretable information describing the intended processing as much as possible.

PG #7: The architecture should not be directly visible to users of the environment.

-- this enhances the freedom to modify the architecture

PG #8: The architecture should be based upon a central information repository.

-- even if it is not implemented as a single centralized utility, a conceptually unified information repository simplifies the problems of adding tool or information fragments to an environment

-- it should be possible for developers of tools to access information fragments as if they were held in a central repository, even if the actual implementation is a distributed database

The remaining principles and guidelines concern properties that the process of developing an architecture for an environment should possess if the resulting architecture is to facilitate extensibility.

PG #9: The tool fragments and information fragments should be defined with the intent of multiple use in realizing the environment capabilities.

PG #10: The tools and information structures provided by the environment interface should be highly decomposed in developing the tool and information fragments defined by the architecture.

PG #11: Definition of the tool and information fragments should take into account fragments that have already been defined in other environment design efforts.

PG #12: Requirements and/or specifications for the environment should indicate possible enhancements so that likely extensions can be anticipated and allowed for in design of the architecture.

-- this is akin to planning for upward compatibility

PG #13: The definition of the tool and information fragments should be guided by the ability to automatically re-generate new versions.

-- developers should consider already known re-generation techniques

-- they should also develop new techniques for regeneration

PG #14: Decisions on tool implementations and data representations should be deferred as long as possible.

-- delayed binding time will decrease the probability that early decisions will make later modifications difficult or impossible

3.3. Alternative Architectures for Extensible Environments

Presenting detailed examples of architectures is premature and beyond the scope of this study. It is, however, instructive to consider broad classes of architectures which might be employed in an environment and the relationships among those classes. This provides a starting point for considering the problem of comparing and evaluating candidate architectures for an environment.

In considering possible classes of architectures, we will look at several that are not layered. This might seem to contradict our guidelines, enunciated in the preceding section, that endorse layering. But, it should be noted that almost all of the environments covered in the survey from which those guidelines were derived had layered architectures, which led to the inclusion of a guideline favoring layering in that list. A list of candidate architectures, generated from first principles rather than from a survey of existing environments, could be expected to violate some or all of the guidelines that arose from cataloging prior experience, and that is precisely what happened in the case of the layering guideline.

3.3.1. Some Classes of Architectures

In this section we describe four classes of architectures. Since an architecture is a conceptual organization for an environment, what we present here are four different conceptualizations for how an environment might be organized. Each conceptualization, or general architectural class, might have any number of realizations or implementations. A single implementation might also be viewed from more than one conceptual perspective and hence represent more than one specific class. Our interest in this section is solely in the conceptual view -- the architecture -- so implementation issues will not be considered further here.

The first class of architectures to be considered may be characterized in terms of 'conditions' and 'triggering'. That is, the conceptual view of an environment whose architecture is in this class is that events are triggered, or occur, in the environment when certain conditions hold. For example, given the conditions 'source program present' and 'executable version requested', a 'compilation' event might be triggered. Similarly, the condition 'test run completed' might cause the event 'test results displayed' to occur.

A second class of architectures can be described as federations of objects, where some or all of the objects are instances of abstract types. In this conceptual view, an environment is composed of groups (federations) of objects, and the operations that can be performed on, or by, at least some of those objects are constrained to be within a predefined set. The predefined set of operations is just those prescribed in the definition of the abstract type to which the object belongs (i.e., of which the object is an instance).

A third class of architectures may be characterized as a collection of concurrent processes communicating with one another via messages. Environments whose architectures fall into this class are explicitly assumed to admit the possibility of concurrent operation among their component parts, although an environment in which no concurrent activity in fact occurred could perfectly well be viewed from this conceptual perspective.

A final class of architectures may be described as a layered organization of tool and information fragments. The layers in such an architecture may be based on any of a number of relationships. For example, the fragments in one layer might define a virtual machine which is used to implement the layer above it. In this case the upper layer would be related to the lower layer by the 'implemented using' relationship (i.e., a fragment in the upper layer is 'implemented using' one or more fragments from the lower layer) while the lower layer would be related to the upper layer by the 'used in implementing' relationship (i.e., a fragment in the lower layer is 'used in implementing' one or more fragments in the upper layer).

3.3.2. Relationships Among Classes of Architectures

As a first step toward comparing and evaluating classes of architectures, it is necessary to characterize the architectures and the relationships among them. A complete characterization is beyond the scope of this study, but in this section we suggest a starting point for characterizing the relationships among the sample classes of architectures listed in the preceding section.

One dimension along which classes of architectures might be compared is the way in which the interrelationships among their constituent parts are constrained. In terms of the examples considered in the preceding section, we can distinguish three distinct points along this dimension:

-- unconstrained: neither the conditions and triggering architecture class nor the concurrent parts and messages architecture class implied any particular constraints on the interrelationships among their constituent parts

-- federated (or grouped): some amount of grouping is imposed by the architecture, as in the federation of objects architecture class described in the preceding section

-- hierarchical: a grouping and one or more relationships among the groups is imposed by the architecture, as in the layered architecture class considered in the preceding section

Another dimension along which classes of architectures might be classified is the type of interactions among their components. Here again we can distinguish three points along this dimension based on our example classes of architectures:

-- procedure call invocation: active components in both the federated and the layered classes of architectures interact via procedure calls, and synchronous message passing among concurrent processes (e.g., Ada rendezvous) may also be considered to be an instance of this type of interaction

-- triggering: active components in the conditions and triggering class of architectures interact indirectly, with conditions caused by one potentially leading to triggering of another

-- asynchronous message passing: active components in architectures from the concurrent parts and message passing class may interact through this mechanism, in which the sender of a message need not await receipt of that message by some recipient before continuing with its work

Another way in which classes of architecture might be compared is according to how highly structured architectures in the various classes are. The four example classes given in the preceding section can be said to be listed in increasing order based on how highly structured they are. The conditions and triggering class of architectures can be considered the least structured, since architectures in that class are described entirely in terms of a list of conditions and triggers. The class of federation architectures can be considered to consist of more structured architectures, since constituents in the architectures of this class have been grouped in some fashion. Architectures in the class characterized by concurrent parts and message passing are yet more structured, since constraints on the type of interactions have been imposed. Finally, the layered class of architectures might be considered to be the most structured of the four classes, due to the constraints on the organization of those architectures imposed by the relationships defining the layers.

3.4. Comparison/Evaluation Criteria

It would obviously be desirable to be able to objectively evaluate architectures or classes of architectures with respect to their support for extensibility. Developing criteria by which such an evaluation can be carried out will require more effort than could be brought to bear in the current sudy, however. In this section we provide some suggestions and starting points that may eventually lead to the development of such criteria.

Several ways of evaluating the extensibily properties of an architecture or class of architectures might be considered. These include:

-- a figure of merit for each architecture or class; this would produce an absolute ranking on which comparisons could easily be made

-- relative comparisons of architectures or classes; this would not give absolute rankings but would allow any given pair of architectures or classes to be compared and hence could yield a favored choice

-- determination that a given class of architectures did or did not facilitate extensibility; this binary clasification scheme would only yield a set of acceptable choices for extensibility, not a clear choice

It seems that the central problem to be addressed in establishing any of these comparison methods is to delineate a suitable set of characteristics and relationships that can be used to characterize environment architectures. Given such a set, some scale or metric for evaluating the characteristics and relationships would then be needed in order to permit a comparison of the architectures as a whole to be performed.

Our preliminary effort at delineating relationships among classes of environments, as contained in the preceding section, illustrates the complexity of this problem. First, it is difficult simpluitable set of characteristics. The dimensions of interrelationships and interactions both seem significant, as does the characterization based on how highly structured an architecture is. Yet these two seemingly related characterizations do not have any obvious well-defined relationship to one another. Moreover, even given a set of characteristics or relationships, it is not clear how to assign values for an architecture based on that set. One can argue, for example, that a more highly structured architecture facilitates extensibility, on the grounds that it limits the possible ramifications of modifications, or that a less highly structured architecture is better for extensibility, on the grounds that modifications require less effort. Similarly, assigning a figure of merit to a property such as layering (Are three layers too few? Are eight too many?) is problematic.

The conclusion is that comparison and evaluation criteria for extensibility of environment architectures is an area deserving further study, preferably of an empirical nature. While it seems unlikely that an absolute ranking mechanism will emerge, at least in the near term, it should be possible to do a significantly better job of evaluating architecture extensibility properties than is possible at present.

3.5. Tradeoffs with Other Architectural Characteristics

Even assuming that the extensibility properties of environment architectures could be accurately assessed, it may not be appropriate to select an environment architecture based solely on that assessment. Tradeoffs between extensibility and other properties of an environment architecture will need to be considered. Here we discuss several such tradeoffs that seem particularly significant.

The impact of extensibility on the extent of integration in an environment must be a prime consideration. Certain approaches to facilitating extensibility, such as maximizing independence and isolation among the components of an environment's architecture, could tend to reduce integration. Given the importance of integration to the productivity of software developers, increased extensibility might well not be worth the price of decreased environment integration. While there is no reason to believe that extensibility and integration are necessarily antithetical, the impact that specific approaches to extensibility may have on integration bears careful consideration.

Efficiency may also be negatively affected by certain properties of environment architectures that are intended to increase extensibility. As was noted earlier, for example, a layered architecture may result in a significant efficiency penalty if not carefully implemented. Here again, there is no obvious reason that efficiency must suffer drastically if an architecture is to provide good support for extensibility, but the potential tradeoff must be kept in mind.

A final possible tradeoff is between support for extensibility and the verifiability properties of an environment architecture. It is possible, for example, that an architecture which makes an environment easier to modify will make it more difficult to prove secure. While this seems unlikely, it probably merits some attention.

3.6. Relationship with Other Architectural Design Issues

The topic of extensibility is related to several other architectural design issues being considered in other studies like this one. Those relationships are briefly considered here.

Architectures that support good human engineering share several features with architectures facilitating extensibility. A central tenet of the human engineering study is that human engineering requires a great deal of malleability in an environment. This makes human engineering and extensibility natural allies. The suggestion from the human engineering study that an environment's architecture should segregate those components handling the user interface is in harmony with the extensibility principle that favors layered architectures.

Similarly, the human engineering considerations favoring a r prototyping capability also argue for the provision of tools supporting automatic generation of environment components, which is a desirable feature from the perspective of extensibility as well. Finally, the human engineering principle that favors a separation of concerns tends to support an environment architecture in which information controlling the environment's behavior is encoded in a form interpretable by the environment itself, which once again is a seemingly valuable attribute for extensibility.

Distributed computing and extensibility are also related. Isolation of parts, which may be a valuable architectural trait for extensibility, should facilitate distributed computing. As noted previously, however, the provision of a conceptually centralized data repository, for extensibility reasons, may impose some added requirements on the kinds of distributed environment architectures that would be acceptable.

There is also a clear relationship between the topics of extensibility and information interfaces, since well-defined interfaces, and the technology to support this property, are evidently an important contributor to facilitating extensibility. Conversely, an architecture with good extensibility properties is likely to impose some significant restrictions on the types of information interfaces found in an environment.

Methodology support and extensibility are also linked, since the primary motivation for an architecture facilitating extensibility is to allow for methodological changes. In particular, by circumscribing a set of potentially acceptable methodologies one might well restrict the extent and direction of possible extensions to an environment, which would clearly have major implications for extensibility.

Finally, as noted previously, extensibility and multi-level security may be related, to the extent that support for extensibility impedes verifiability of an environment.

4. Strategies for Creating and Evolving Extensible Environments

Extending an environment is inherently a process of change. One way to cope with the difficulty of such a process is to design an infrastructure for the environment that is impervious to change. Such an structure would be able to accommodate changes to the complement of tool and information fragments without having to change itself.

It is improbable that one can develop such an ideal infrastructure. Doing so would require infinite insight into the future so that the infrastructure was developed with complete knowledge of both the nature and the details of possible future changes. It would also require development of a infrastructure having a level of generality that is probably beyond our ability to create and would certainly have a degraded, and possibly unacceptable, level of performance.

The attributes of architectural infrastructures that are highly, but not necessarily totally, impervious to environment interface changes was, in essence, the subject of the previous section. In this section, we turn to the complementary topic of establishing a growth pattern for environments that simplifies and eases the process of extending them. In our discussion, we include, but do not make special recognition of, changing an environment's architectural infrastructure when that proves necessary to accommodate changes to the environment's functionality.

4.1. Impact of Extensibility on Environment Creation and Evolution

In any process of decision making, early decisions establish a context for later ones, shaping the issues that must be subsequently addressed and constraining the options available for resolution of an issue. One effect is that changing an earlier decision can have a relatively far reaching impact. It can potentially invalidate many of the later decisions, and it can open the door to new options that were not previously considered. Another effect is the importance of trying to place early in the process those decisions that will not severely, and certainly not overly, constrain the options available for resolution of later issues. Preserving a large number of options can increase the quality of later decisions because of the greater variety of options available. It can also increase the probability that later decisions can be changed without having to remake earlier ones in order to provide a richer set of options.

Creating and evolving an environment is a decision making process. Creating and evolving an extensible environment is a decision making process which must accommodate change as easily as possible. As such the process should adhere to three principles suggested by the previous discussion:

> Principle of Late Binding: aspects of the environment that are likely to change should be fixed as late as possible
>
> Principle of Option Preservation: aspects of the environment that are fixed early should minimally constrain the options available at later points in the process
>
> Principle of Decreasing Generality: aspects of the environment that are fixed early should be as general as possible but the environment's aspects fixed late in the process need not have a high level of generality

In the rest of this subsection, we will address the general impact of trying to adhere to these principles. In the next two subsections, we then discuss the impact on initial creation and subsequent evolution, respectively. In the final subsection, we will indicate tools that can enhance our ability to extend an environment over time.

To be able to adhere to any of these principles, we must know at least the general nature, if not the specific details, of changes that will likely be requested. This suggests that we should expand the scope of the requirements definition phase beyond merely specifying the capabilities to be provided to include the specification of likely enhancements. For any sort of environment, this means that the specification of capabilities supporting specific activities must include information about how these capabilities might be expanded or changed. In the case of comprehensive, modern environments, there is the additional implication that possible shifts to alternative paradigms for creating and evolving software should be detailed in the environment's specification.

Knowing about changes that might be requested will only partially assist in ordering the decision making process so that the above principles may be observed. In addition, we will need the ability to determine tradeoffs among alternative orderings and the ability to determine the impact of decisions on the options available for other decisions. This suggests that investigation of existing efforts, to learn about tradeoffs and the option-restriction characteristics of various creation and evolution scenarios, must be actively done in the early, requirements definition, phase of developing an environment. It also suggests that empirical studies will be needed to fill knowledge gaps stemming from the lack of direct, existing experience. It is important that these investigations and studies address the issue of decision ordering rather than be designed to merely garner the technical details of various options for the decisions that will be encountered during environment creation and evolution.

Knowing about future enhancements and determining how to best order decisions to address tradeoffs and preserve options are intuitively desirable. They are also a recognized part of modern approaches to software creation and evolution. When we consider the case of structuring an environment as a hierarchy of virtual machines, however, we arrive at a suggestion that is both counter-intuitive and somewhat at odds with modern practice.

Recall that a hierarchical virtual machine structure enhances extensibility by providing 'invariants' -- the lower level machines -- that can be preserved in the face of change. Relating this to the principles stated above, this means that the lower level machines should reflect decisions made early in the environment creation and evolution process. This leads to the suggestion that environment creation and evolution should progress bottom-up through the virtual machine hierarchy.

This is really a specific instance of a more general suggestion: incorporate mechanisms that support a variety of options and speed up the process of making a change once it has been decided upon. General, powerful virtual machines are one such mechanism. Table-driven language processors are another. All such mechanisms support adhering to all of the principles cited above and the principle of option preservation in particular.

The overall process should therefore include extensive determination of future enhancements, investigation of extant approaches, empirical study of new alternatives, and early attention to identifying and designing general primitive mechanisms. When considered all together, these suggestions point to use of a prototyping methodology for creation and evolution of extensible environments. Investigation and empirical study are almost synonymous with prototyping. Prototyping is generally recognized as valuable for the clarification of requirements and the determination of future enhancements is just an extension of requirements clarification And prototyping offers a way to cope with the dilemma of having to fix lower level aspects of a hierarchically structured system early in the decision making process by forcing the construction of admittedly immature versions that then serve as empirical studies supporting the creation of later versions.

Prototyping also offers the possibility of begging the major issue of what should be done/decided first. With enough study and investigation, it might be possible to determine the "correct" order of decision making for extensibility -- in fact, had the study reported on here been allotted more than two months for completion, we may have come up with more extensive suggestions for specific architectures or specific creation and evolution scenarios. But the preparation of comprehensive, modern environments is a high-risk activity with little guiding experience -- and it is all the more so with the additional requirement that the result exhibit a high level of extensibility. Prototyping is the best, currently known approach to coping with this risk and uncertainty [5] and its effect in terms of foreshortening up-front planning and gedanken investigation seems fully warranted after our relatively short investigation of environment extensibility.

4.2. Scenarios for Initial Creation

Taken to the extreme, prototyping reduces initial creation of an environment to a position of relative insignificance -- the really important step is evolving the next version from the previous one. Nonetheless, there are some relatively important decisions that must be made initially before any versions are constructed. These decisions concern the general nature of the environment's structure and the way in which it delivers its capabilities. In this section, we indicate what some of these decisions should concern -- -- good and bad outcomes for the decisions have been discussed earlier. To make our comments more specific, we focus attention, in this and the next subsection, on environments that are structured as a hierarchy of virtual machines.

One issue of primary concern should be the rough decomposition of the environment into a hierarchy of successively more primitive facilities. This will form the basis for defining and designing the environment's virtual machine layers. It will also help establish an overall "game plan" for carrying out the prototyping because it will help in deciding whether the prototyping should progress truly bottom-up through the virtual machine layers or be a more broad-band approach in which successive versions capture some of all the layers.

Making this rough decomposition amounts to determining decompositions for the environment's tool and information fragments. It will, therefore, require addressing the following issues:

-- what will be the conceptual basis underlying the tool fragments, for example, will they all view the software modeled as communicating sequential processes?

-- what is the structural model for the documents available through the environment's interface, for example, will the documents be viewed as having a book-like structure with chapters, sections, subsections, paragraphs, and sentences?

-- what is the environment user's view, for example, will users think in terms of traditional activities such as compilation, linking, etc.?

-- what sequences of activities will environment users perform, for example, will they usually perform a data flow analysis before execution but sometimes skip this activity?

-- which already available virtual machines (for example, CAIS, the procedure "machine" in the FASP environment, or the information repository "machine" from Joseph) should be used?

-- which generators, such as the syntax-directed editor generator from Gandalf, should be employed?

Some decisions about these issues will follow from the definition of the individual capabilities required of an environment and the possible future changes to these capabilities. Other decisions will follow from considering the scenarios under which the capabilities will be used, which is to say the general methodologies to be supported by the environment over its life time. Still others will follow from considering the technology that is available or on the horizon.

Another issue of primary concern should be the general "game plan" for later evolution of the environment. This requires addressing the issues such as the following:

-- what will be the general schedule for releasing versions of the environment to its users and what will be included in each release?

-- should successive prototypes work up through the virtual machine layers, should each capture some parts of all layers, or should there be a combination of these extremes?

Decisions about these issues will establish the general structure for the environment and the general framework underlying its extension in the future. These decisions will also establish the general growth pattern through which the environment will be evolved and extended.

4.3. Scenarios for Evolving Environments

Evolving an environment structured as a hierarchy of virtual machines is relatively straightforward. At each step, about the only decision is whether to localize modifications to one of the layers or make a more lateral move in which several layers are modified. This will be affected by what has been released as well as the nature of the required change. The point of making the overall approach a prototyping one is that change will be the norm rather than the exception and can therefore be driven more by concerns for minimal impact on the users and the scope of the required change rather than by concerns for the impact of the change on the environment's structure or the technical feasibility of actually making the change.

The scenario for evolving an extensible environment will be impacted by whether extensions can only be made by a single organization or whether anyone who has the environment will be able to extend it. Even if it is a policy that only a single organization can make changes, it is likely that (unapproved) extensions will be made by other organizations. In addition, allowing any organization to extend the environment will increase the probability that an extension has already been made somewhere in the community at large when the requirement for this extension arises. It is perhaps best, therefore, to establish a scenario for evolution that does not localize responsibility for extension but rather fosters extension by all members of the community and establishes the mechanisms for propagating extensions throughout the community and certifying that extensions adhere to quality standards and rules for preserving the environment's basic structure and underlying framework. (Such a scenario has been defined in Appendix 6 of a recent study of programmatic alternatives for the STARS Program [9].)

4.4. Automated Support for Creating and Extending Environments

In general, software tool support is needed to support a prototyping approach that uses standards and guidelines to preserve commonality across a broad community of users. Specific support is needed to be able to investigate tradeoffs, conduct empirical studies, discern the impact of proposed changes, assess the validity of changes, and determine where changes have to be made in order to accomplish some intent. Software tools supporting these activities are discussed in this section under several broad categories.

4.4.1. Instrumentation and Monitoring Tools.

Guidance for changing the environment's interface and responding to required changes to the interface requires data about what capabilities are being used and how they are used. An extreme example would be being able to, as a side effect of a required change, completely change some aspect of the environment's interface without ill effect if it were known that this aspect had not been used by any of the users. A more normal example would be having performance related information that would help guide the reorganization of the environment should that ever become necessary to extend it.

To gather this information, the environment must be instrumented to collect information about the patterns of activity carried out by the users and the internal processing needed in response to environment use. Exactly what data should be collected is, of course, determined by what issues must be addressed. Rarely is it completely known what data will be needed, so a not-so-unusual practice is to collect as much data as possible at a fairly low level of granularity, approaching (and sometimes actually achieving) the level of keystrokes during interactive use. This obviously will impact the performance of the environment itself and so every effort is needed to focus the data collection on that which is really needed and an appropriate level of data granularity.

Tools to automatically instrument a software system have been developed -- the ones available in the Toolpack environment [8] are representative. These usually place data probes under the guidance of statements made at the source level for the system. They are also usually oriented towards collecting the data needed for system testing, but it should be possible to easily adapt them for environment extensibility purposes.

Data collection is only half the picture and the ability is needed to analyze the data to discover distributions of statistics and make inferences about use scenarios as well as patterns of internal activity. Reduction of the data to statistical distributions is straightforward and can be accomplished by the use of any of a variety of existing tools, for example, the ones included in the DCDS environment. Tools supporting the analysis of data to infer patterns of activity are generally not available. There were some early attempts to provide tools that tried to infer environment usage patterns by analyzing a keystroke history, but significant advancement in this area has been lacking.

4.4.2. Configuration Management and Version Control Tools.

Any large-scale software development effort that extends over a long period of time will need support for management of the various configurations that arise and control of the numerous versions that will exist. Prototyping approaches to this development will only make the need more severe.

Configuration management and version control tools are widely available. All of the current Government-sponsored APSE implementation efforts will, for example, include such tools in the resulting environments. And most modern operating systems, Unix for example, include such tools. In addition, several firms, among them Softool in Santa Barbara [26], provide standalone configuration management and version control systems programmed in higher level languages.

It is uncertain how well traditional tools will assist grappling with the configuration management and version control problems that arise during prototyping. The much larger number of versions and the higher frequency of change will certainly stress most tools to, and perhaps past, their limit. About the only way to address this issue is to try some traditional tools in prototyping situations and see how they hold up and where, if at all, they are insufficient.

4.4.3. Tool and Information Fragment Decomposition Tools

Assistance will be needed to decompose information and tool fragments to fit into the environment's infrastructure. To the extent that it will be necessary to account for fragments that already exist in the environment or elsewhere, this will largely be a manual process that will rely on experience and general knowledge. Some help for tool decomposition could be provided by module "discovery" tools such as found in the Argus environment [27] -- these would help identify tightly interrelated portions of the system that are candidates for tool fragments. Extensive, high-quality assistance will have to await more experience or some direct, research attack on the problem.

4.4.4. Generic Tool Piece Parts.

Generic, reusable tool piece parts have been extensively developed for compiler technology. General, table-driven lexical analyzers and parsers have existed for over a decade and advances have recently been made in extending the same concept into the arena of compiler back-ends. These will be of high utility in preparing extensible environments because of the basic need to process the descriptions that users prepare. Their utility will be even more extensive if the use of languages is extended to describe more about an evolving software system than just its operation.

Other tools provided by an environment have recently been the target of this style of implementation. For example, the syntaxdirected editors provided within Gandalf environments are, for the large part, reusable processors that work on specific languages through the use of tables describing the languages. It can be expected that more tools will be structured in this way as they are more fully understood and it becomes more apparent what information can be captured declaratively as opposed to imperatively within the tool's code itself.

Another aspect of providing generic tools is to identify primitive piece parts that are of general utility in a large number of situations. This has been done somewhat within the Unix system -- many of its tools perform simple tasks and can effectively be used in a wide variety of tools. Of particular value in this regard are generic tool piece parts that perform text manipulation -- a software engineer recently observed to one of the authors that he used the egrep string matching tool in Unix to either completely or partially do over 70% of the activities he had to perform.

As noted above, effective use of generic tools requires mechanisms for their efficient use. Thus, additional "tools" such as the Unix piping facility and efficient mechanisms for driving tools with tables of data will be required.

4.4.5. Tool Libraries.

As generic, reusable tool piece parts proliferate, tool libraries will be needed to help organize them and make it easy to identify what is available for specific situations. Standard library facilities can be used to organize a collection of tool fragments. More will be needed to catalogue them so that they can be retrieved according to their dynamic as well as their static characteristics. This problem has been discussed in [4] with regard to the development of libraries for reusable software.

4.4.6. Tool Building Tools.

Currently, support for building tools is limited to the capabilities discussed above with respect to generic tool piece parts. For example, there are generators that create the tables used to drive generic parsers, lexical analyzers, compiler back-ends, and syntaxdirected editors.

It is difficult to identify what more is needed until we have more experience with building extensible environments and other software systems that require the rapid development of new tools or variations of existing tools. This experience will help identify processing that can be provided in a generic form. It will also help identify techniques for generating specific instances of the generic tools.

4.4.7. Analysis Tools.

Any software creation and evolution effort requires the ability to analyze interactions among modules for suitability. This requires analysis for correct use of an interface such as determining that the types of arguments agree with the types of the corresponding parameters. Tools for this sort of analysis are commonly available, often as part of a compiler but frequently as separate tools as in the DCDS environment. But as interface definition becomes more complex, for example as a result of the PIC and Interface Contract work discussed above, more extensive capabilities will be required.

Analysis of module interaction also requires the ability to assess the dynamics of interface usage over time. Most current capabilities of this sort rely upon some of sort data flow analysis, for example the dynamics analysis tools developed for the Arcturus environment [28]. But alternative approaches, such as provided in the TOPD environment [24], have also been developed. More work is required before these tools provide all the functionality that is needed and exhibit acceptable performance characteristics.

4.4.8. Standards Enforcement Tools.

Standards will play a role in environment extensibility in order to support tool transportability and data interoperability. They will also be necessary to help control the unrestricted proliferation of tools and modification of the environment should responsibility for extensibility not be vested in one organization as suggested above. To the extent that standards are defined, there will be a need to enforce adherence to the standards by checking that they are observed.

Some standards will be very easy to check, for example standards having to do with the form of information interfaces. Standards that pertain to temporal characteristics, such as the appropriate usage of tool piece parts over time, will be considerably more difficult to check. The ability to check observance of standards will, for the most part, co-evolve with the ability to perform analysis that was discussed in the last subsection.

4.4.9. Prototyping Support Tools.

All of the tool capabilities discussed above support a prototyping approach to extensible environment creation and evolution. In addition, variants of these tools or special tools will be needed to support the basic philosophy of prototyping, namely gradual maturation through the frequent preparation of new versions which provide small extensions to previous versions. This will largely affect the operational characteristics of the tools that are used. It will also require tools that assist in quickly preparing new versions such as tools that help identify where changes should be made and help install to quickly install those changes.

An example of the additional sort of tool that will be needed is the make facility in Unix for the generation of versions. Again, more experience is needed to determine what is needed beyond current capabilities.

4.4.10. Summary of Tool Support

Many of the tools needed to support a prototyping approach to the creation and evolution of extensible environments exist today. It will be a major task to collect these tools together into an integrated environment supporting the development of extensible environments. This integration will be complicated by the fact that the environment for building an extensible environment is most effectively the extensible environment itself since we have suggested that the community using the extensible environment be encouraged to extend it.

Enhancement of the tool set to something providing extensive assistance must await more knowledge, gained through experience, concerning where the current capabilities are deficient.

5. Conclusions

We have collected a great deal of information on extensibility as evidenced by existing environments. And we have made some preliminary inferences about general principles and guidelines, future work needed to design an infrastructure for an extensible environment, and generally beneficial characteristics for the "game plan" used to create and evolve extensible environments. We have not had the luxury of time to distill our conclusions one level higher and identify the 'half dozen most important things to do for achieving extensibility'.

We had a preconceived notion: to support extensibility one should develop a highly flexible infrastructure that allows maximal freedom in changing the elements of the tool set and the information repository. We found nothing that contradicted this notion. And we identified a large variety of ways to accomplish this end.

The conclusions that stand out strongest in our minds are the following:

-- use a hierarchical virtual machine structure (but we caution, again, that this may be an artifact of the fact that the existing environments we surveyed all had this structure to at least some degree)

-- support a logical view of the information repository that allows attributes to be associated with information fragments and associations among the fragments to be relatively arbitrarily constructed; do not depend on being able to predefine all attributes and associations and prepare and provide for user-definition of new ones

-- use a prototyping approach to create and evolve the environment; do not rely on prototyping just to determine the requirements, but rather have the production of successively more mature versions be the basic underlying theme of the process

-- develop a collection of tools supporting prototyping and include this collection of tools in the extensible environment itself

-- in preparing the collection of tools, look to using declarative information and generators; make a concerted effort to consolidate tools of this sort that exist today and a determined effort to extend the range of capability of these tools

REFERENCES

1. Ada Joint Program Office. Common APSE Interface Set Definition. Ada Joint Program Office, Department of Defense, October 1984.

2. M. Alford. DCDS Environment Architecture Description. 45674G950-002R1, TRW Defense Systems Group, Huntsville, Alabama, December 1985. (JSSEE Report Number JSSEE-ARCH-005.)

3. B. Bailey. Human Engineering Impact on the Stars SEE Architecture, Institute for Defense Analyses Paper P-1818, April 1985.

4. J. Batz, P. Cohen, S. Redwine, Jr., and J. Rice. The applications-specific area. Computer, 16, 11 (November 1983), 78-85.

5. B. Boehm, T. Gray and T. Seewaldt. Prototyping versus specifying: A multiproject experiment. IEEE Trans. on Software Engineering, SE-10, 3 (May 1984), 290-302.

6. J. Buxton and L. Druffel. Requirements for an Ada programming support environment: Rationale for Stoneman. Proc. IEEE Compsac Conf., Chicago, October 1980, pp. 66-72.

7. G. Clemm. ODIN An Extensible Software Environment; Report and User's Reference Manual. Technical Report CU-CS-262-84, Department of Computer Science, University of Colorado, 1984.

8. W. Cowell and L. Osterweil. The Toolpack/IST programming environment. Proc. SoftFair Conference, Arlington, Virginia, July 1983.

9. R. DeMillo, A. Marmor-Squires, W. Riddle and S. Redwine, Jr. Software Engineering Environments for Mission Critical Applications -- Alternative Programmatic Approaches. IDA Paper P-1789, Institute for Defense Analyses, Alexandria, Virginia, August 1984.

10. A. Evans, Jr. and K. Butler (eds.). Diana Reference Manual (Revision 3). Technical Report TL 83-4, Tartan Laboratories Inc., Pittsburgh, Pennsylvania, February 1983.

11. C. Green, D. Luckham, R. Balzer, T. Cheatham and C. Rich. Report on a Knowledge-Based Software Assistant. Technical Report RADCTR-83-195, Rome Air Development Center, Griffis Air Force Base, New York, August 1983.

12. H. Hart et al. Report on a Workshop on Future APSE Environments. To appear: Software Engineering Notes, April 1985.

13. Intermetrics. Architectural Description of the Ada Integrated Environment (AIE). IR-MA-423, Intermetrics Incorporated, Cambridge, Massachusetts, December 1984. (JSSEE Report Number JSSEE-ARCH-003.)

14. G. Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, Reading, Massachusetts, 1983.

15. D. Lamb. Sharing Intermediate Representations: The Interface Description Language. Technical Report CMU-CS-83-129, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1983.

16. M. Lipczynski. Software Architecture of the Facility for Automated Software Production. Advanced Software Technology Division, Naval Air Development Center, Warmister, Pennsylvania, December 1984. (JSSEE Report Number JSSEE-ARCH-004.)

17. R. Mitze. The Unix system as a software engineering environment. In: Hunke (ed.), Software Engineering Environments, North-Holland Pub. Co., Amsterdam, 1981.

18. Naval Underwater Systems Center. Architectural Description of the Ada Language System/Navy (ALS/N). Report 3511, Naval Underwater Systems Center, Newport, Rhode Island, December 1984. (JSSEE Report Number JSSEE-ARCH-002.)

19. J. Nestor, W. Wulf and D. Lamb. IDL Interface Description Language: Formal Description. Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, February 1983.

20. D. Notkin et al. Special Issue on Gandalf. J. Systems and Software, to appear May 1985.

21. L. Osterweil. Toolpack -- An Experimental Software Development Environment Research Project. IEEE Transactions on Software Engineering, November 1983, pp. 673-685.

22. W. Riddle. The evolutionary approach to building the Joseph software development environment. Proc. SoftFair Conference, Arlington, Virginia, July 1983.

23. T. Standish. A Philosophy for a Tool Extension Paradigm. Computer Science Dept., University of California, Irvine, August 1982.

24. R. Snowdon and P. Henderson. The TOPD system for computer-aided system development. In: Bergland and Gordon, Tutorial: Software Design Strategies, IEEE Computer Society, 1979.

25. SofTech. Architectural Description of the Ada Language System

(ALS). SofTech, Middleton, Rhode Island, December 1984. (JSSEE Report Number JSSEE-ARCH-001.)

26. Softool. Applicability of Softool's Change and Configuration Control Environment to the DoD Software Life Cycle. Softool Corp., Santa Barbara, California, n.d.

27. L. Stucki. What about CAD/CAM for software? The ARGUS concept. Proc. SoftFair Conference, Arlington, Virginia, July 1983.

28. R. Taylor. A general-purpose algorithm for analyzing concurrent programs. Comm. ACM, 26, 5 (May 1983), 362-376.

29. R. Taylor and T. Standish. Steps to an advanced Ada programming environment. Proc. Seventh Intern. Conf. on Software Engineering, Orlando, Florida, March 1984, pp. 116-125.

30. A. Wolf, L. Clarke and J. Wileden. Ada-Based Support for Programming-in-the-Large. IEEE Software, 2, 2 (March 1985), 5871.

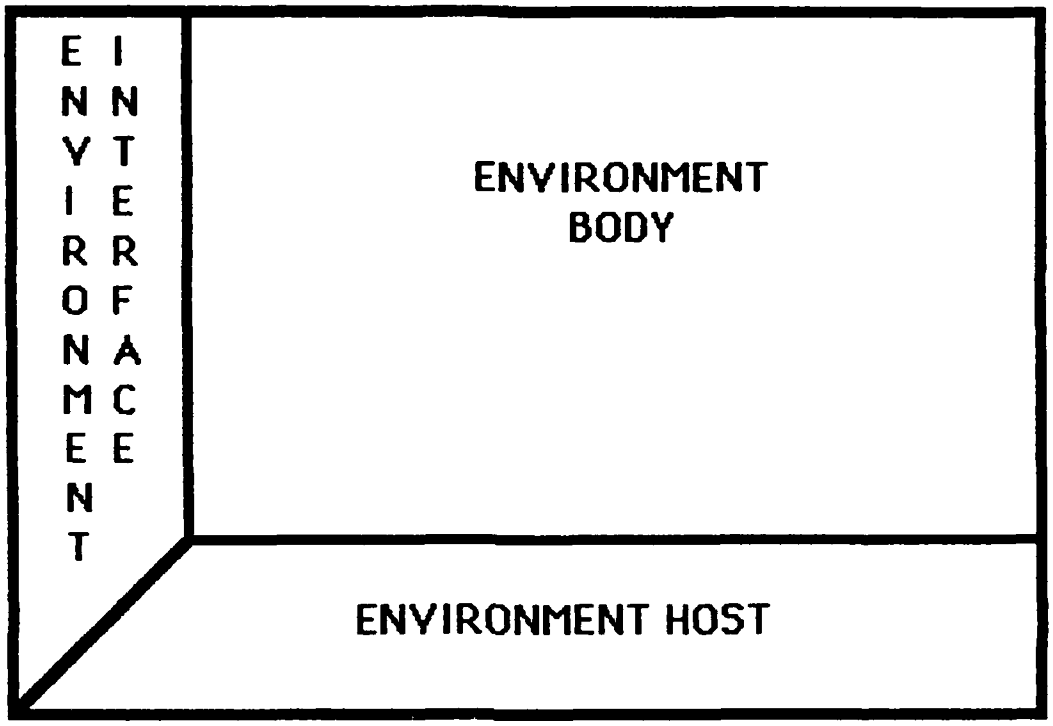


Figure 1: A View of a Software Engineering Environment

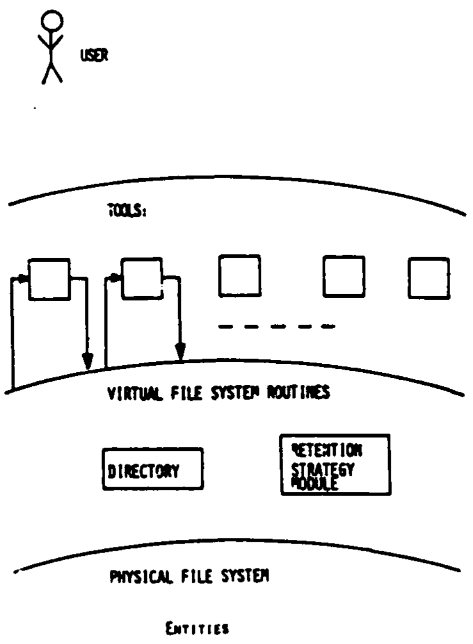


Figure 2: Toolpack Architecture

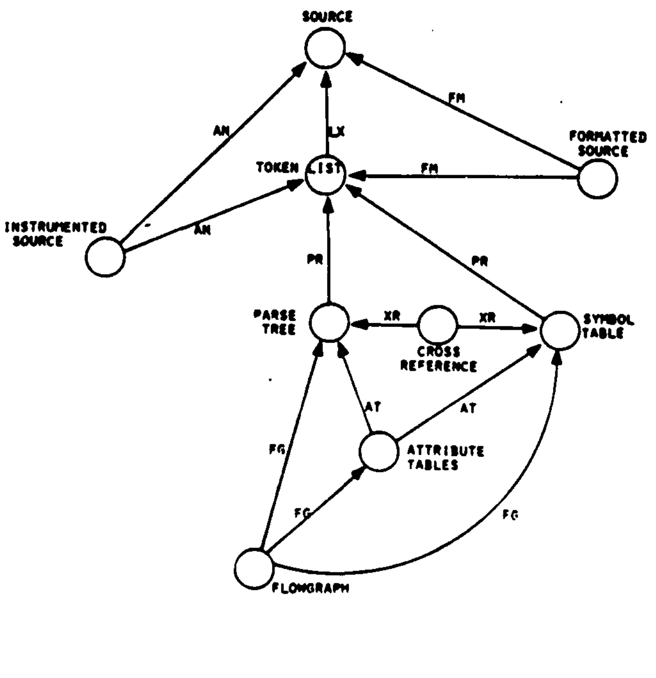


Figure 3: Example Toolpack Dependency Graph

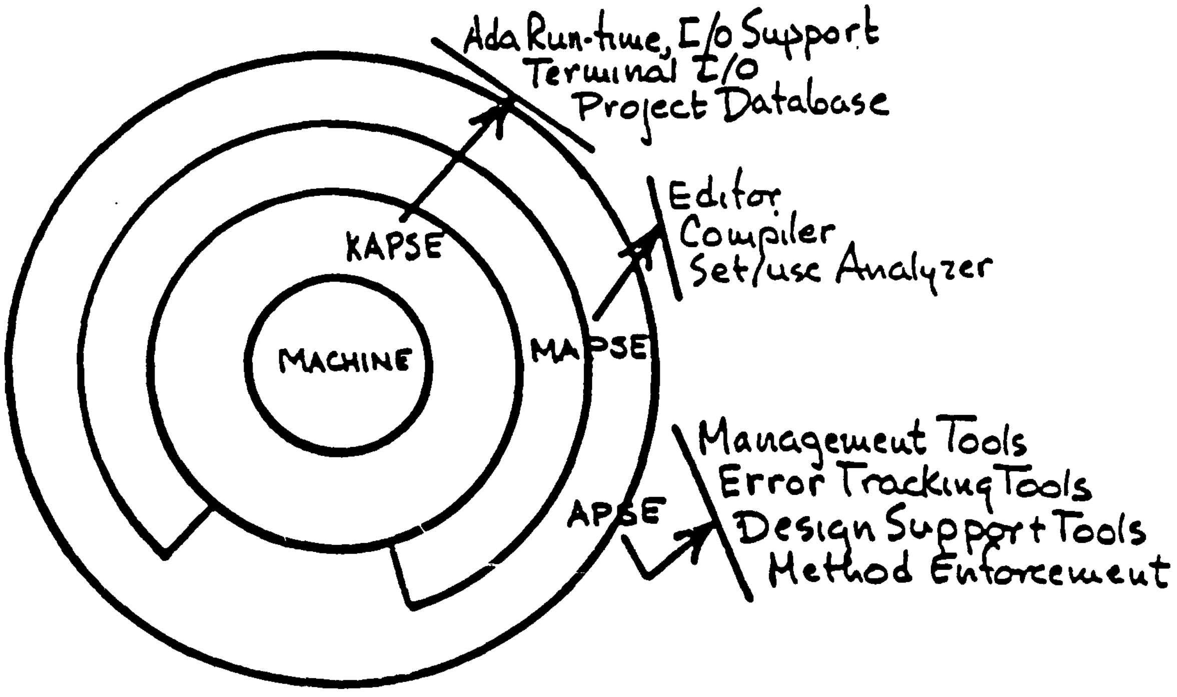


Figure 4: Stoneman APSE Architecture

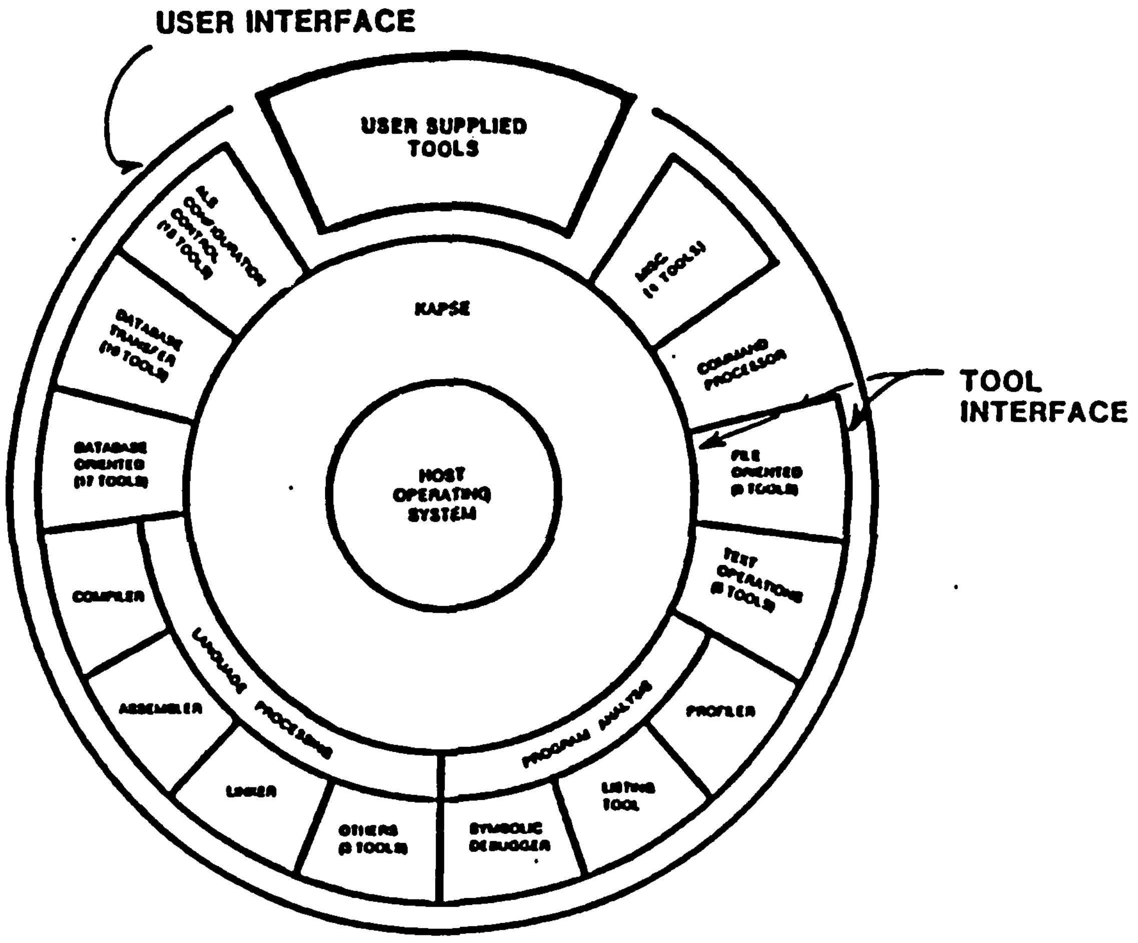


Figure 5: Ada Language System Architecture

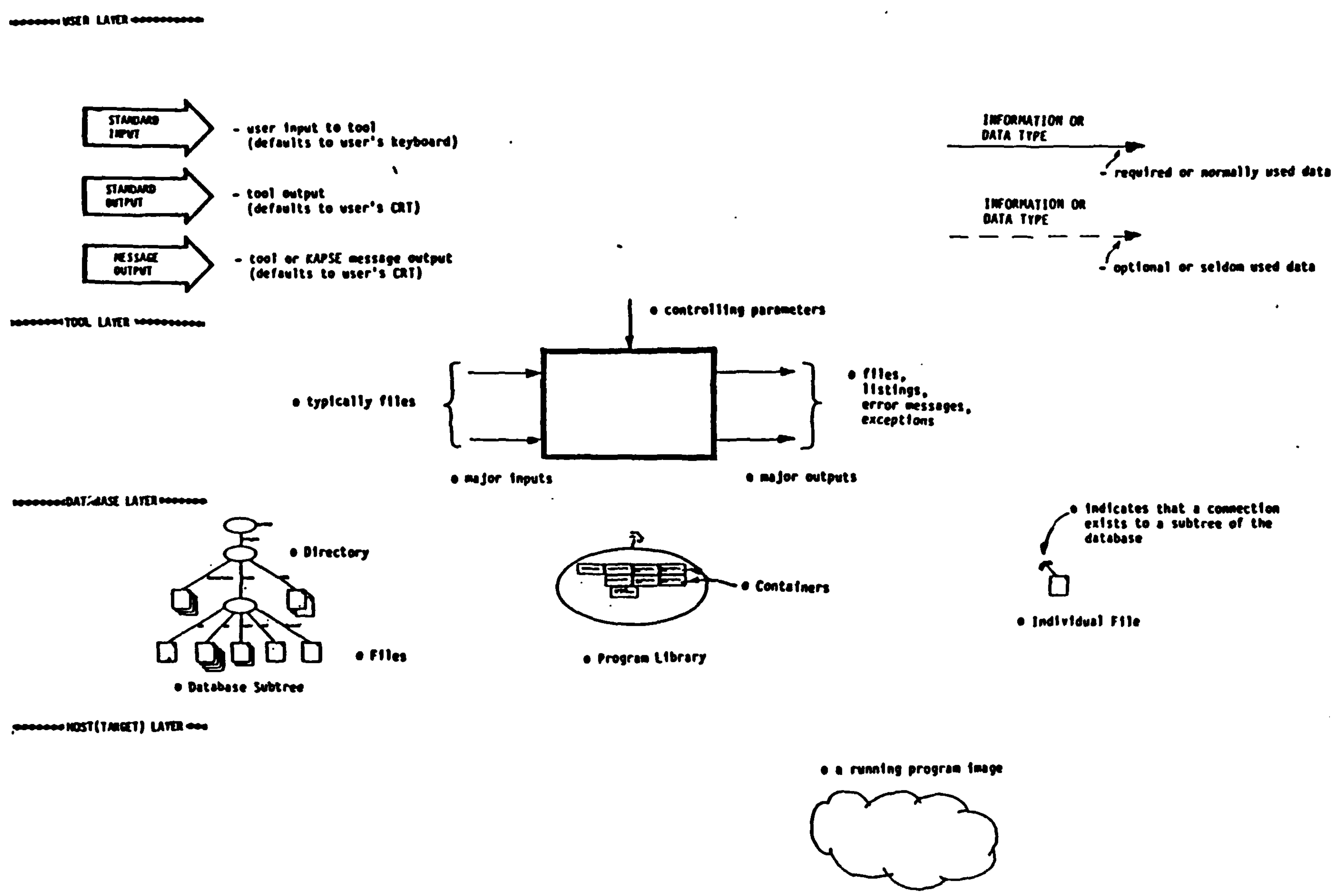

USER LAYER
STANDARD INPUT
- user input to tool (defaults to user's keyboard)
STANDARD OUTPUT
- tool output (defaults to user's CRT)
MESSAGE OUTPUT
- tool or KAPSE message output (defaults to user's CRT)
INFORMATION OR DATA TYPE
- required or normally used data
INFORMATION OR DATA TYPE
- optional or seldom used data
TOOL LAYER
• controlling parameters
• typically files
• files, listings, error messages, exceptions
• major inputs
• major outputs
DATABASE LAYER
• Directory
• Files
• Database Subtree
• Containers
• Program Library
• indicates that a connection exists to a subtree of the database
• individual File
HOST(TARGET) LAYER
• a running program image

Figure 6: Model of Ada Language System Dynamics

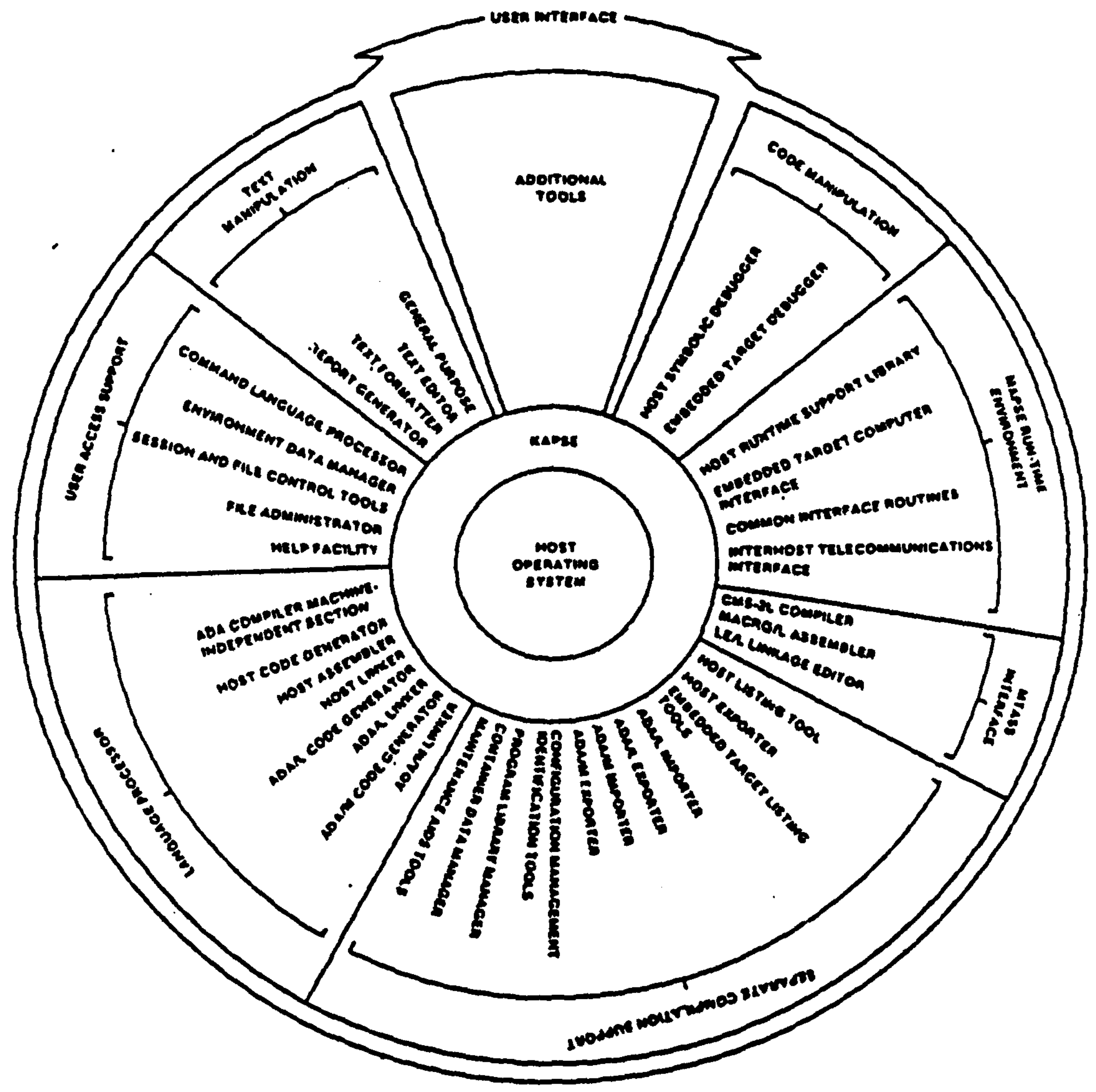


Figure 7: Ada Language System/Navy Architecture

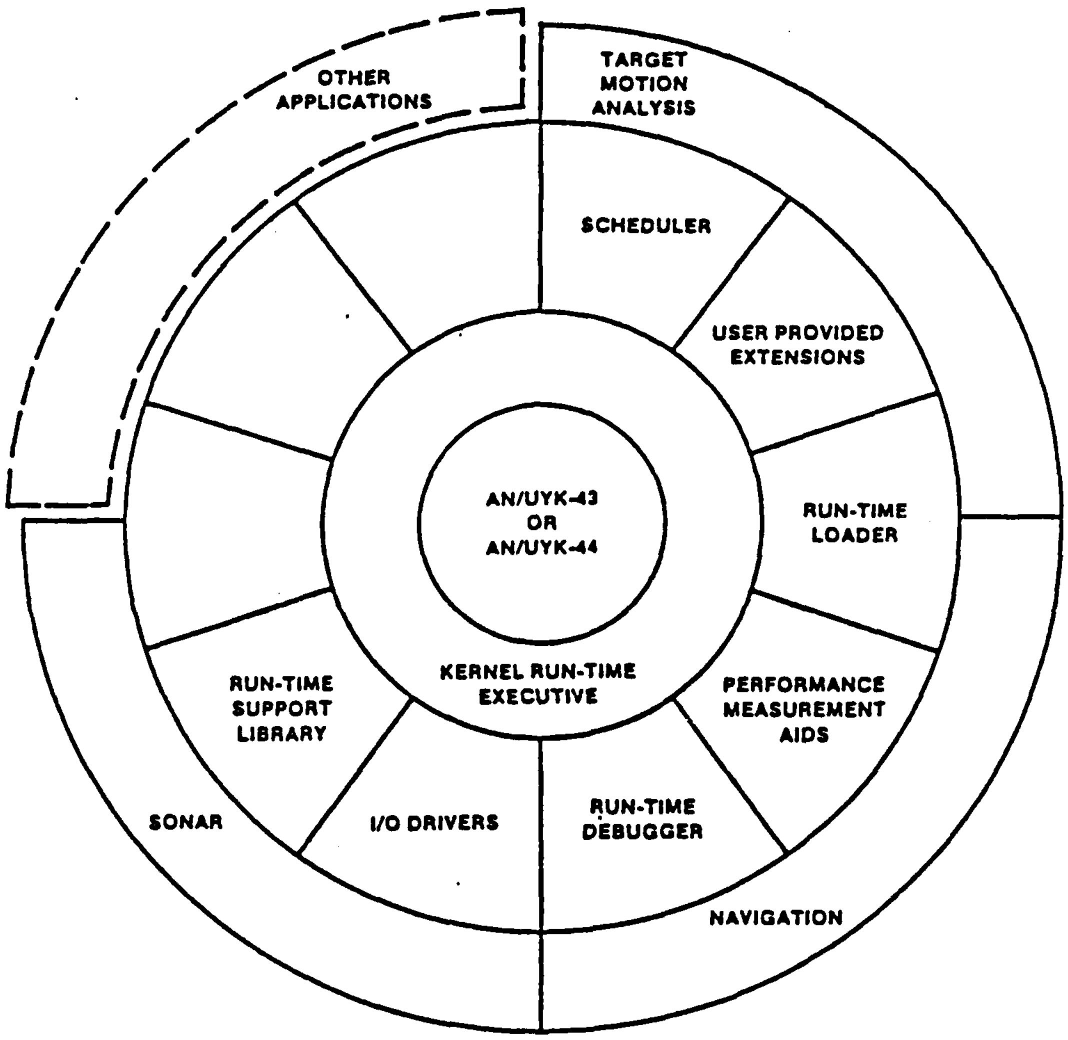


Figure 8: Architecture of the Run-time Executive in the Ada Language System/Navy

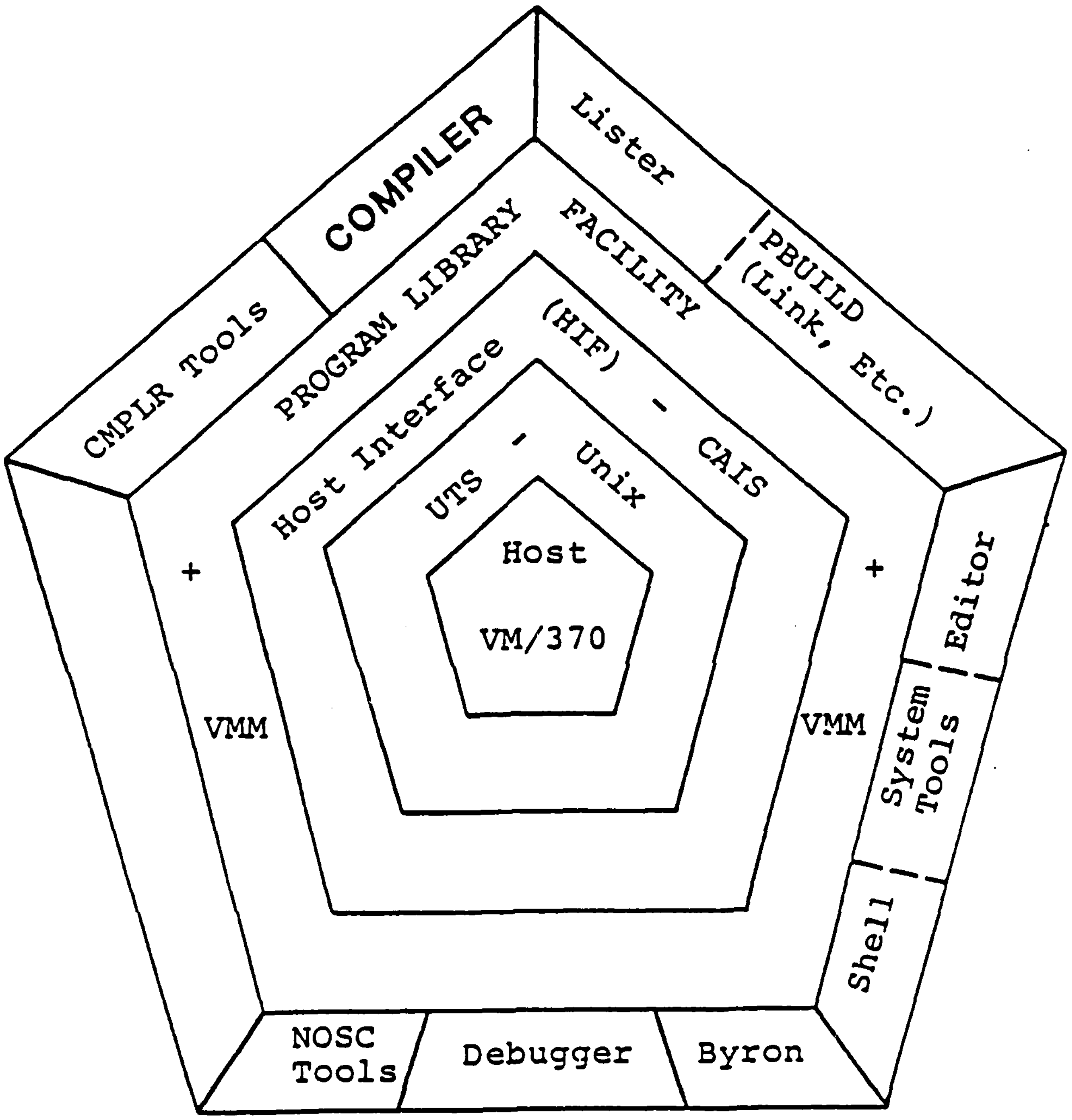


Figure 9: Ada Integrated Environment Architecture

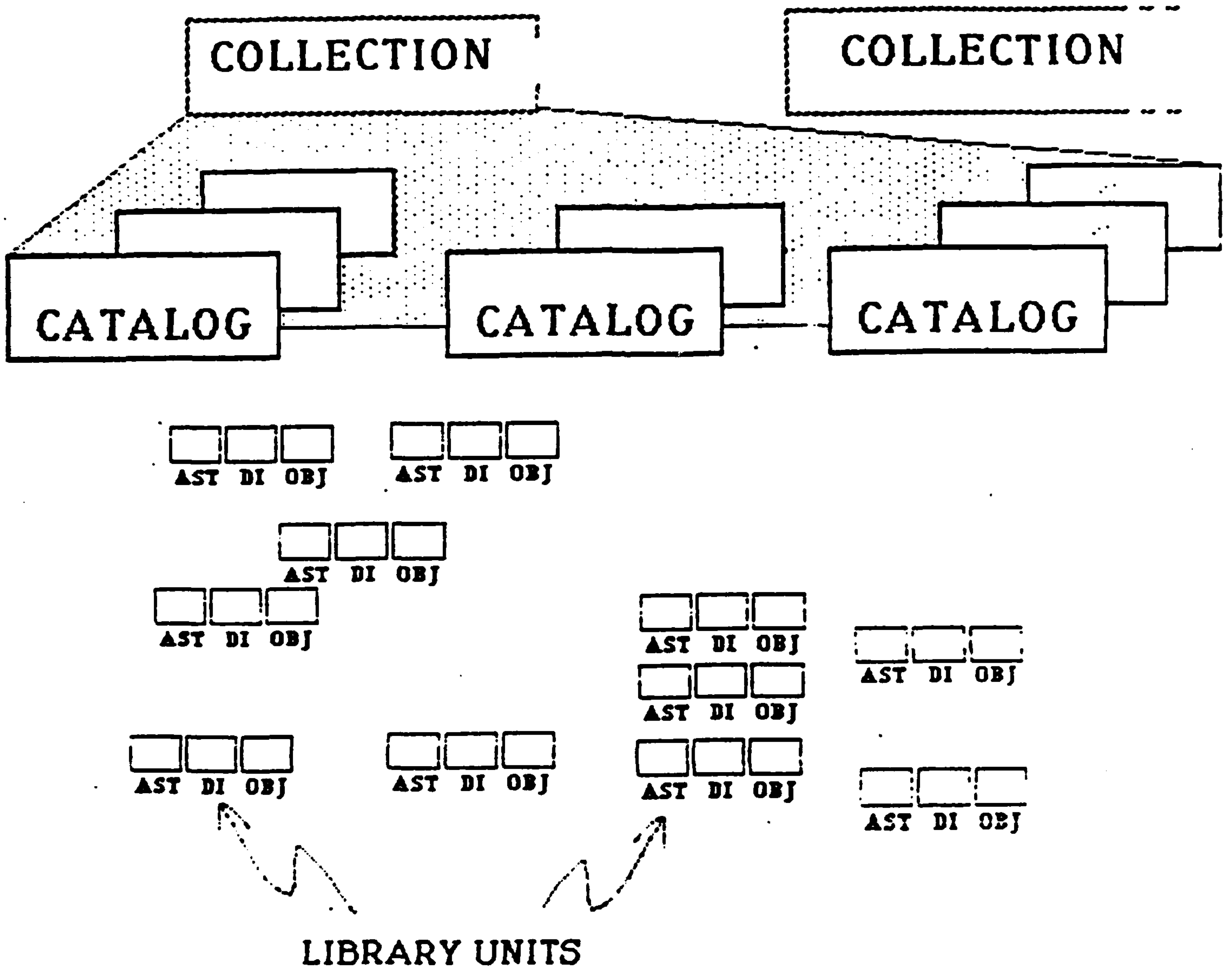
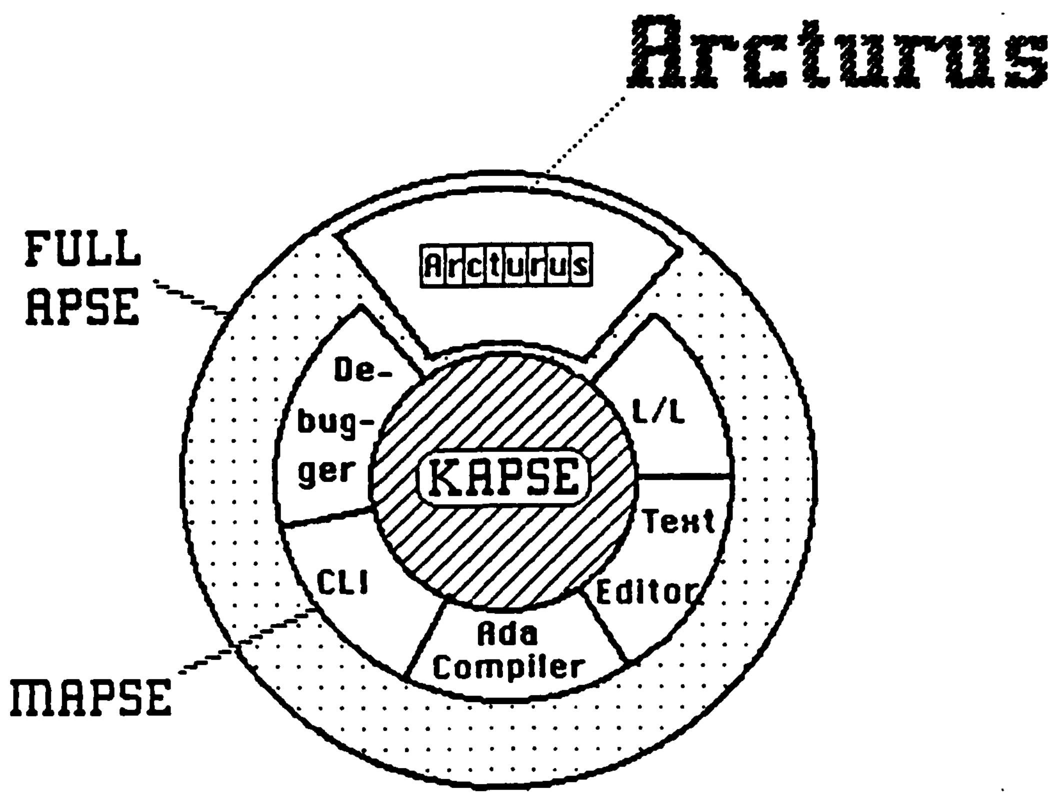


Figure 10: Structure of the Ada Integrated Environment Program Library



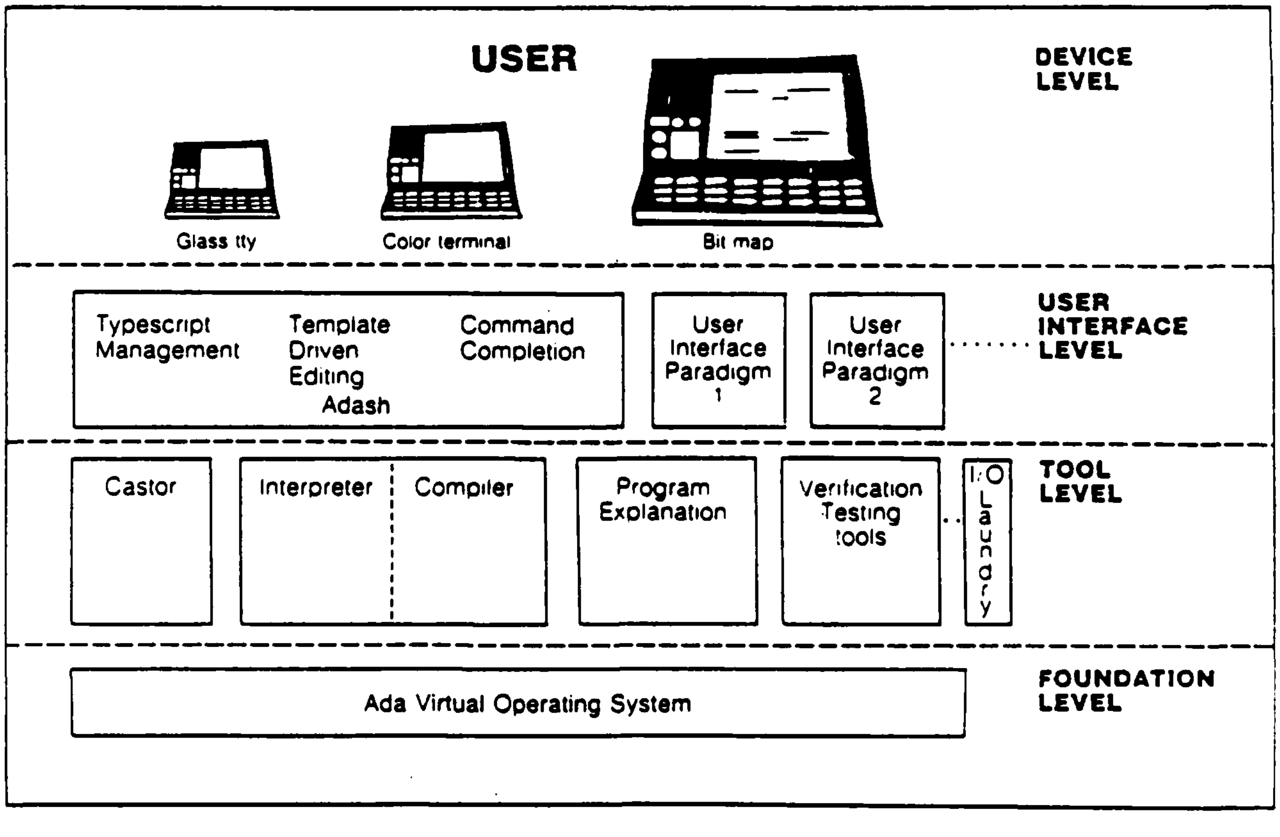


Figure 11: Arcturus Architecture

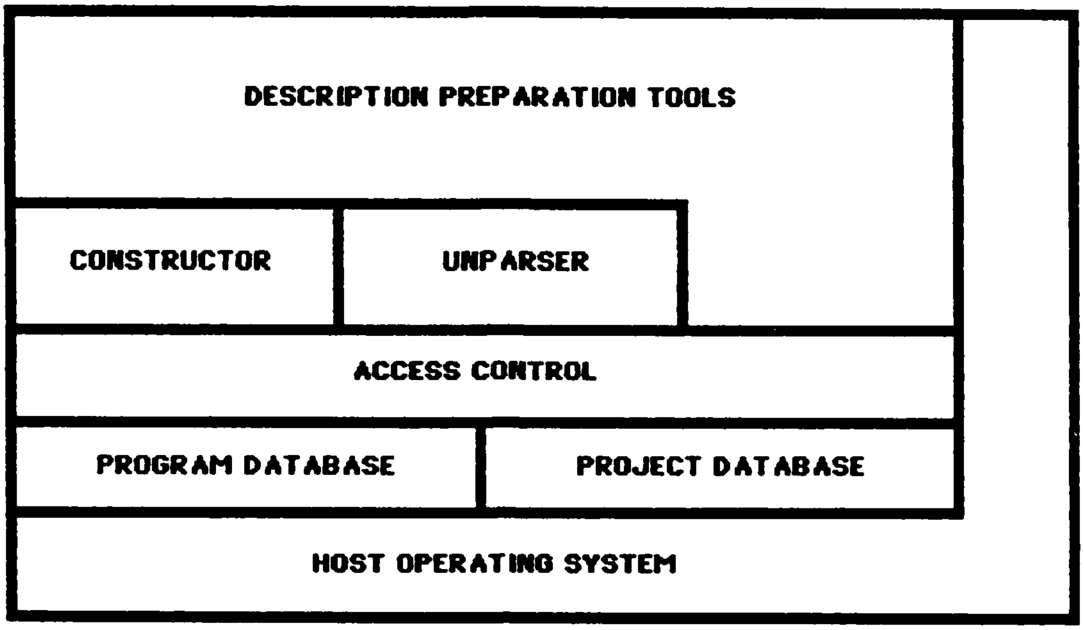


Figure 12: Gandalf Architecture

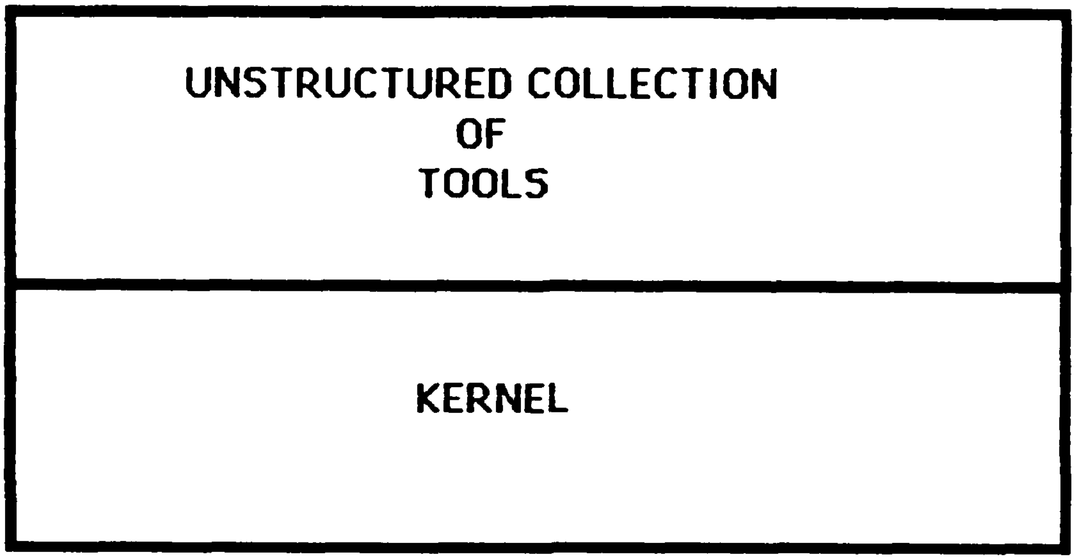


Figure 13: Unix Architecture

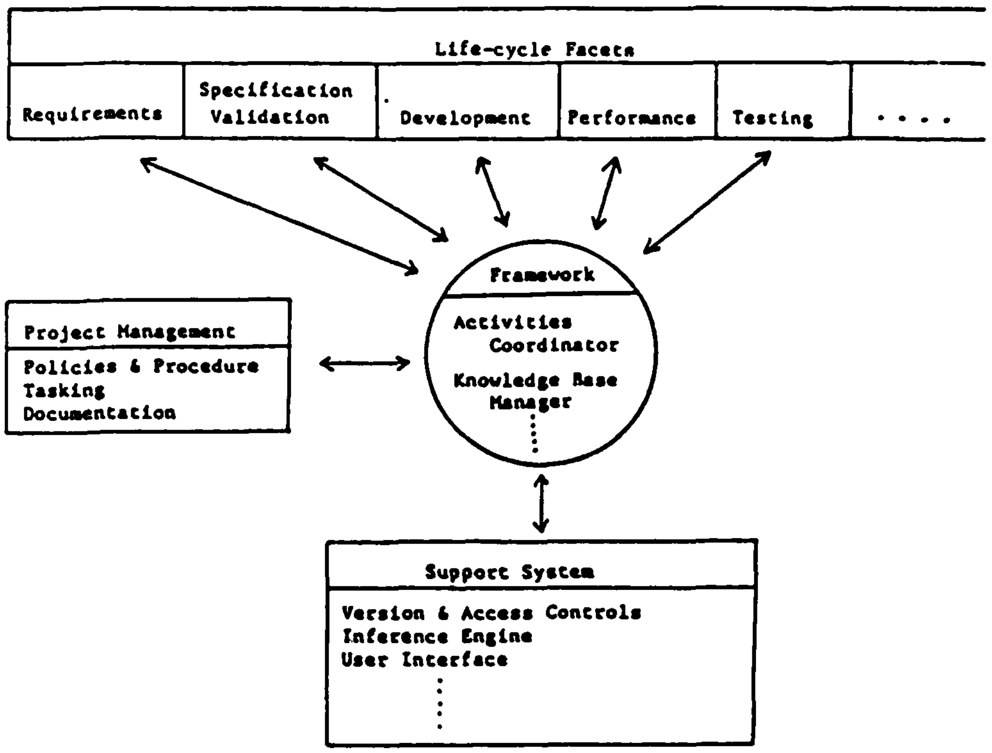


Figure 14: KBSA Architecture

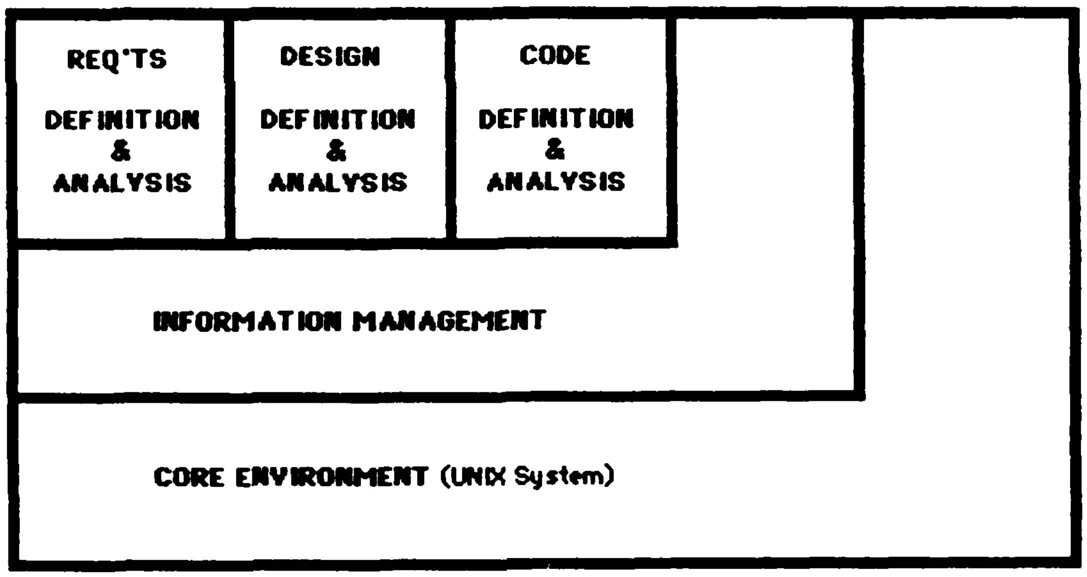


Figure 15: Joseph Architecture

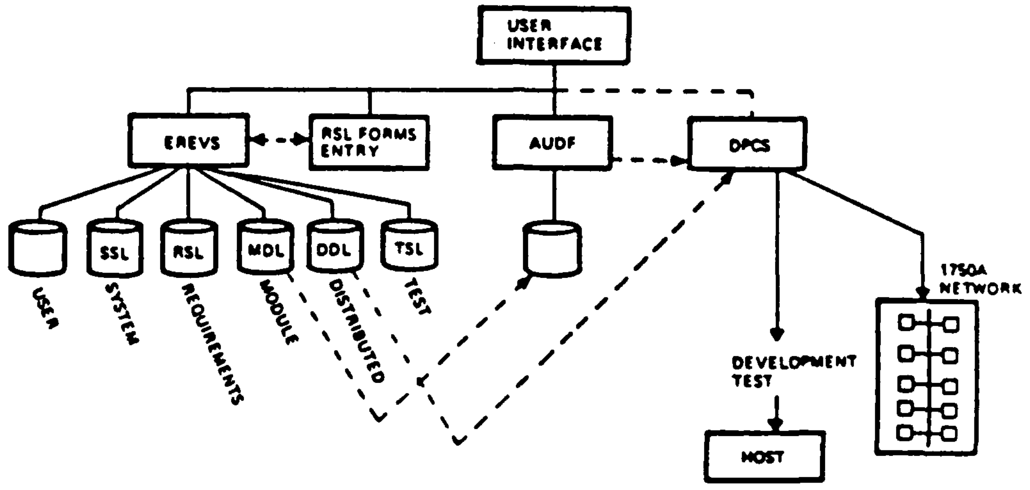


Figure 16: DCDS Architectural View

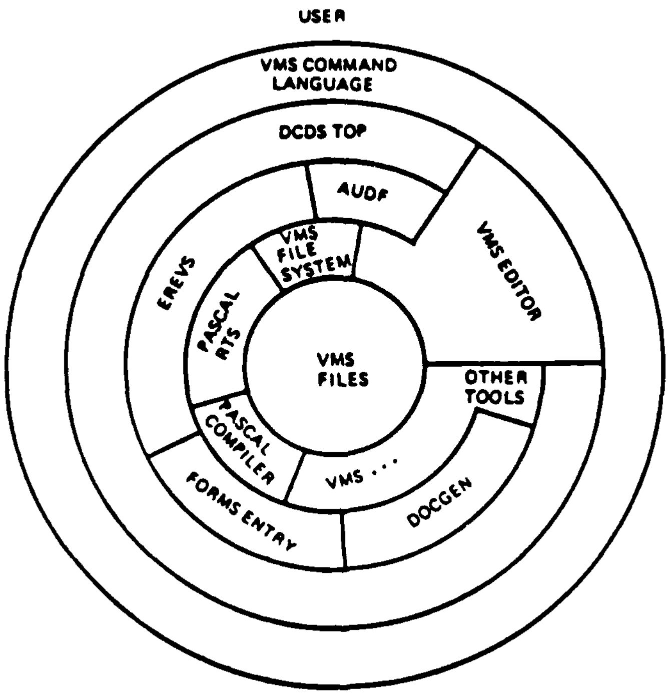


Figure 17: Layered Architecture for DCDS

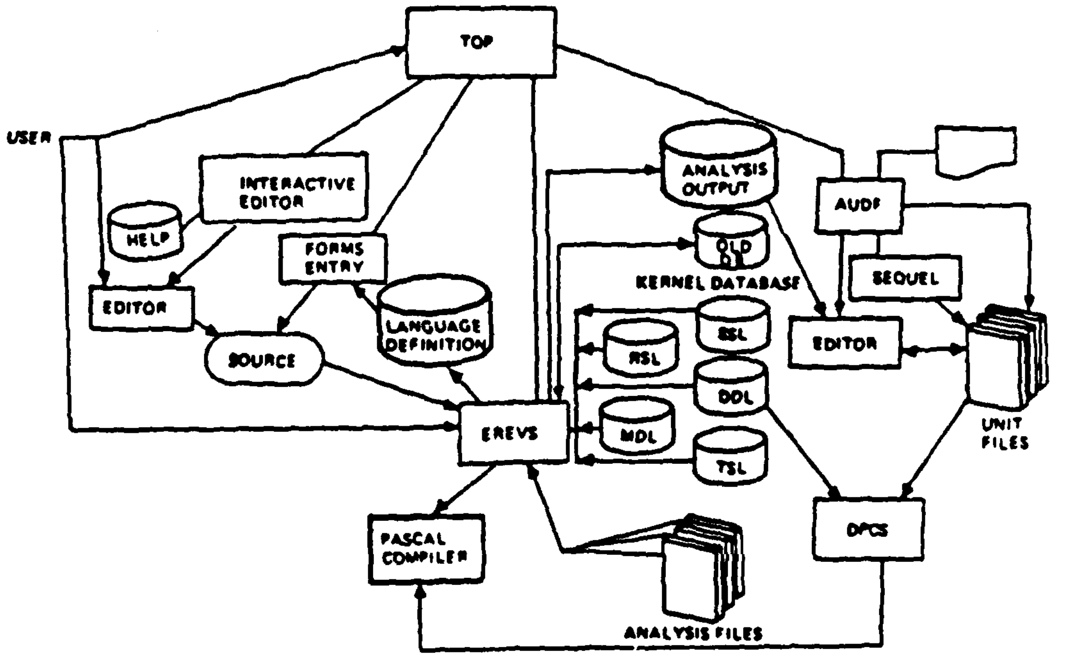


Figure 18: DCDS Block Diagram

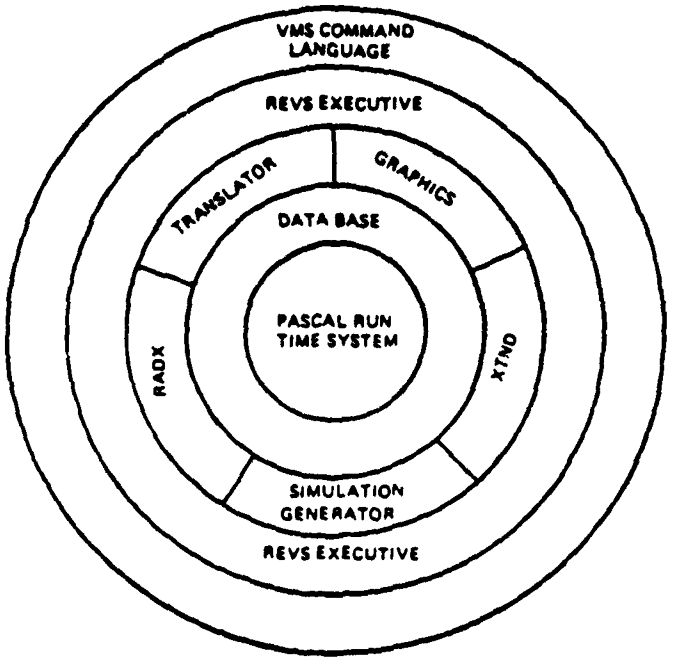


Figure 19: Layered Architecture for EREVS

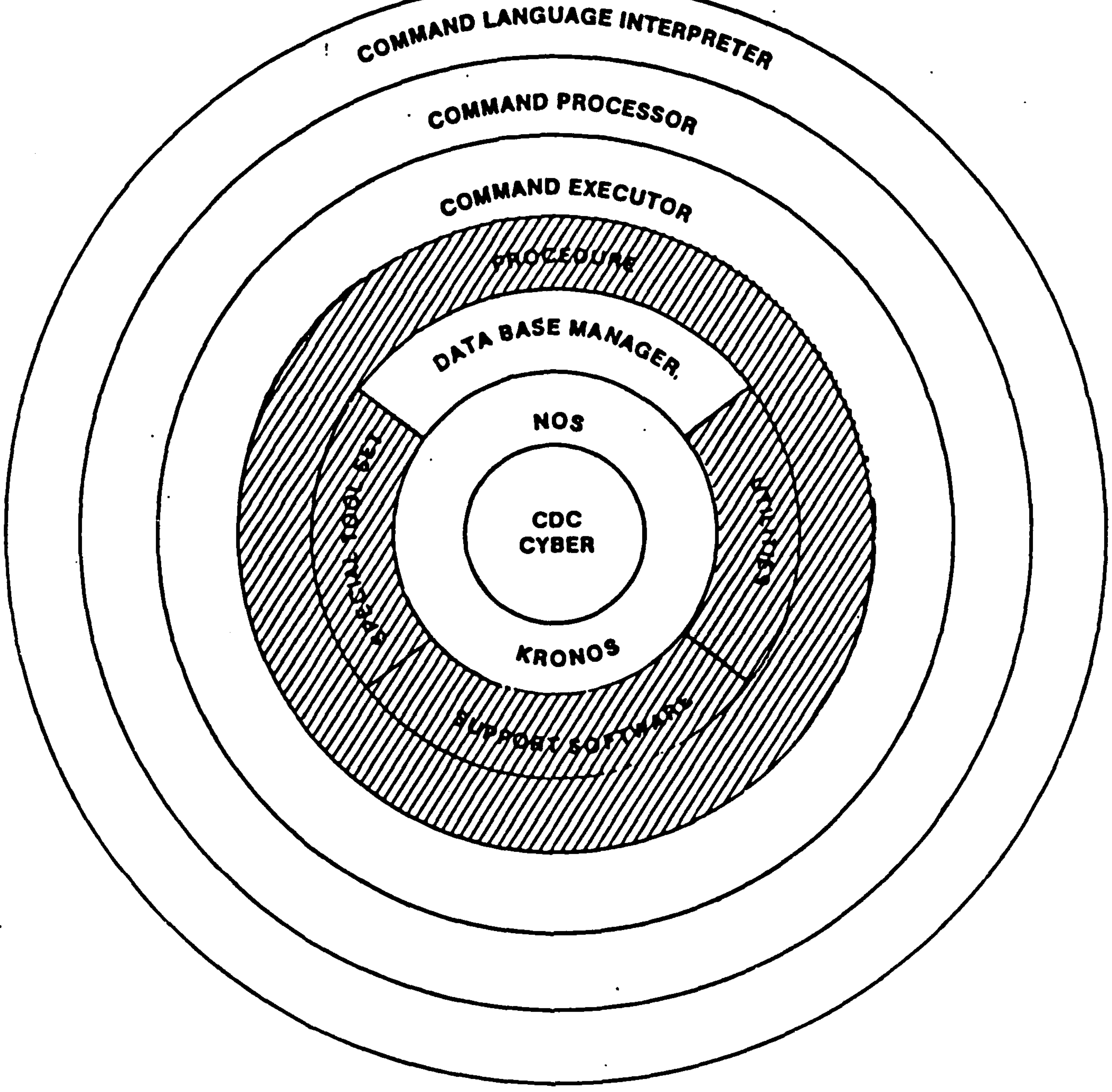


Figure 20: FASP Architecture

Distribution List for P-1828

DoD

Col. Joe Greene Director, STARS Joint Program Office 1211 Fern St., C-107 Arlington, VA 22202	10 copies

Other

Mr. Jack C. Wileden
COINS Dept.
Lederly Graduate Research Center
University of Massachusetts
Amherst, MA 01003

Dr. William Riddle
Software Productivity Consortium
1880 Campus Common Dr., North
Reston, VA 22091

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2 copies

CSED Review Panel

Dr. Dan Alpert, Director
Center for Advanced Study
University of Illinois
912 W. Illinois Street
Urbana, Illinois 61801

Dr. Barry W. Boehm
TRW Defense Systems Group
MS 2-2304
One Space Park
Redondo Beach, CA 90278

Dr. Ruth Davis
The Pymatuning Group, Inc.
2000 N. 15th Street, Suite 707
Arlington, VA 22201

Dr. Larry E. Druffel
Software Engineering Institute
Shadyside Place
480 South Aiken Av.
Pittsburgh, PA 15231

Dr. C.E. Hutchinson, Dean
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

Mr. A.J. Jordano
Manager, Systems & Software
Engineering Headquarters
Federal Systems Division
6600 Rockledge Dr.
Bethesda, MD 20817

Mr. Robert K. Lehto
Mainstay
302 Mill St.
Occoquan, VA 22125

Mr. Oliver Selfridge
45 Percy Road
Lexington, MA 02173

<u>IDA</u>

Gen. W.Y. Smith, HQ	
Mr. Seymour Deitchman, HQ	
Ms. Karen Webber, HQ	
Dr. Jack Kramer, CSED	
Dr. John Salasin, CSED	
Dr. Robert Winner, CSED	
Ms. Katydean Price, CSED	2 copies
IDA C&D Vault	3 copies

END
12-87
DTIC